Low power and high performance heterogeneous computing on FPGAs

Doctoral Dissertation
Doctoral Program in Electronic Engineering (31st cycle)

Low power and high performance heterogeneous computing on FPGAs

Liang Ma
* * * * * *

Supervisor
Prof. Luciano Lavagno, Supervisor

Doctoral Examination Committee:
Prof. A.B., Referee,
Prof. C.D., Referee,
Prof. E.F., Referee,

Politecnico di Torino
February 28, 2019

I hereby declare that, the contents and organisation of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

. . . . . .

Liang Ma
Turin, February 28, 2019

Summary

In the initial decades of the 21st century, we are approaching the ExaScale era in which computing systems will be able to perform up to $2^{60}(\approx 10^{18})$ FLOPS (i.e. floating point operations per second). It is impossible to reach this impressive performance with homogeneous architectures, since high-performance general purpose processors consume too much energy per computation, no matter whether high computational capabilities are required in data centers or in edge applications. Due to the likely end of Moore's law even for scaling, higher and higher demands for low power and high performance are now satisfied only by heterogeneous computing systems, which are composed by host devices and a set of co-processors with various of capabilities to handle particular tasks.

This thesis focus on the design of FPGA-based accelerators, which exploit a reconfigurable spatial computing architecture to achieve massive parallelism at a very low power consumption. The predominant design methodologies for an FPGA is still based on Register Transfer Level (RTL) models, which are entirely diverse from software models and thus prevent the software and hardware co-design and the principle "Write once, run anywhere''. In this thesis we propose several system level design methodologies for FPGAs via high-level synthesis, which enables the portability of software originally written in C, C++ or OpenCL for CPUs or GPUs to hardware platform.

The performance of an application on a piece of hardware is bounded by two factors: the peak performance of the processor/accelerator and the bandwidth of the memory. The roofline model is used to estimate the peak performance of an algorithm according to its computation-to-communication ratio, to classify the algorithm into computation-bounded, memory-bounded or somewhere in the middle and to analyze the bottlenecks of the algorithm for further optimization. Note that the memory bandwidth of FPGAs has been historically lower than that of GPUs, and FPGAs are typically less efficient than GPUs at double-precision floating point, while FPGAs shine when it comes to on-chip memory bandwidth and fine-grained low precision computations.Thus the classification depends on both the algorithms and the devices.

This thesis covers both computation-dominated and memory-dominated designs.

First we consider financial models as examples of computation-intensive algorithms, namely the Black Scholes model and the Heston model of the prices of one vanilla option (i.e. European vanilla option) and two exotic options (i.e. Asian option and European

barrier option) respectively. We optimized and implemented these algorithms on three platform types (i.e. CPUs, GPUs and FPGAs) and a total of five devices. Obviously both FPGAs and GPUs outperformed CPUs significantly. Even an embedded FPGA achieved about 15x better performance than a data center class CPU. The FPGAs achieved 4x to 5x operations per Watt than a GPU fabricated in the same process (e.g. 16nm and 28nm).

Secondly, we proposed to optimize the memory access bandwidth of memory-bounded algorithms, by using pre-designed C++-based inline caches rather than the tedious on-chip local memory design approach. The latter requires designers to manually exploit the memory access patterns and manage the data movements and synchronizations, the on-chip RAM architecture, the kernel interfaces to access external memory, and functional verification. We applied inline caches with different types and configurations to three algorithms from very different application areas such as machine learning, databases, and computer vision respectively. In summary, our cache implementations improved performance by up to 8x and energy by about 2x with respect to the out-of-the-box unoptimized code, achieving comparable results to the best available manual optimizations of the on-chip memory architecture, while requiring a much shorter design time.

Finally, we explored the design space of several types of machine learning algorithms including convolution neural networks and recurrent neural networks. They are both memory and compute-bounded. These models were first designed and trained in a framework such as Tensorflow and then our automation tool generated self-contained C++ projects that are supported by high level synthesis tools. We proposed a dataflow-based acceleration methodology by which we could migrate our designs on various target FPGAs and achieved excellent performance due to the dramatically reduced number of the external memory accesses. We applied this methodology to two neural networks that we designed targeting an embedded FPGA (for edge computations). The first one is a CNN variant named ShiftShuffleNet. The top-1 accuracy is up to 68.5% and top-5 accuracy is up to 88.2% on ImageNet, despite a very heavy quantization with 4-bit activations and 1-bit weights. We designed a configurable ShiftShuffle block where the dataflow paths are configured by the host processors via a set of micro-instructions to implement the full net on a resource-limited embedded FPGA. We not only achieved a competitive accuracy, but also improved the actual inference speed in terms of frames per second (about 96.5 fps). The second is a recurrent neural network based on an LSTM cell. This network was trained by another project in our lab, and had a very good accuracy compared to other feed-forward neural networks. On the embedded FPGA, it achieved a comparable performance and about 2x operations per Watt as a datacenter-class Xeon CPU.

All these experiments prove that high level synthesis is not only mature for both research and commercial uses, but also excellent in terms of the achieved performance and power with respect to CPUs and GPUs. Hence FPGAs can readily be exploited in heterogeneous computing systems for low power and high performance demands in both data centers and edge applications.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Prof. Luciano Lavagno for his assistance in my study and research these years, for his patience, motivation, enthusiasm, and immense knowledge. Not only his guidance helps me to conquer plenty of troubles in my research, but also his life philosophy and values affect me from many aspects.

I would also like to thank my colleges, also my friends, from both the same university and other part of the world. In the collaborations with the teams from the Xilinx, from the Mentor Graphics and from the UC Berkeley, I not only had my horizon broadened, but also gained friendships. Without their passionate participation and input, these projects could not have had these remarkable achievements.

Finally, I sincerely give my profound gratitude to my parents, to my family, to my girlfriend and to my lord for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

I would like to dedicate this thesis to my loving parents

Contents

List of Tables xi

List of Figures xii

1 Introduction 1
- 1.1 Computing System . . . 2
  - 1.1.1 Moore's Law . . . 2
  - 1.1.2 Heterogeneous Architecture . . . 2
  - 1.1.3 High Level Synthesis . . . 5
- 1.2 Performance Analysis . . . 6
  - 1.2.1 Amdahl's Law . . . 6
  - 1.2.2 Gustafson's law . . . 6
  - 1.2.3 Roofline Model . . . 7
  - 1.2.4 Optimization in HLS . . . 7
  - 1.2.5 Power and Energy . . . 11
- 1.3 Thesis Structure . . . 11

2 Acceleration of Financial Algorithms 13
- 2.1 Financial Options . . . 14
  - 2.1.1 European Vanilla Option . . . 14
  - 2.1.2 European Barrier Option . . . 14
  - 2.1.3 American Option . . . 15
  - 2.1.4 Asian Option . . . 15
- 2.2 Option pricing models . . . 15
  - 2.2.1 Black Scholes Model . . . 15
  - 2.2.2 Heston Model . . . 16
- 2.3 Monte Carlo Method . . . 17
  - 2.3.1 Random Number Generator . . . 18
- 2.4 Related work . . . 19
- 2.5 Implementation and Architecture . . . 19
  - 2.5.1 Optimization . . . 21
  - 2.5.2 Performance Indicator . . . 30

2.6 Results . . . 30
2.6.1 Running on CPUs . . . 31
2.6.2 Acceleration by GPUs . . . 31
2.6.3 AWS F1 FPGA . . . 31
2.6.4 Embedded Platform: ZYNQ 7020 . . . 32

3 Cache-Based Acceleration for Memory Intensive Algorithms 37
3.1 Background . . . 38
3.1.1 Memory-intensive algorithms . . . 38
3.1.2 Cache Related Work . . . 38
3.1.3 Motivations . . . 39
3.2 High Level Cache Design . . . 40
3.2.1 Hardware Design Flow . . . 40
3.2.2 Inline Cache Types . . . 42
3.2.3 Inline Cache Implementation . . . 44
3.3 Applications . . . 47
3.3.1 Matrix Multiplication . . . 48
3.3.2 Lucas-Kanade Algorithm . . . 50
3.3.3 Bitonic Sorting . . . 54

4 Acceleration of Machine Learning Algorithms 61
4.1 Introduction . . . 62
4.1.1 Convolutional neural network . . . 64
4.1.2 Neural Network on FPGAs . . . 67
4.2 Design, Training and Inference Automation . . . 68
4.2.1 Tensorflow . . . 68
4.2.2 Design Flow and Code Generation . . . 69
4.3 Feed-Forward Neural Network . . . 71
4.3.1 Dataflow-Based Acceleration . . . 72
4.3.2 Hardware Architectures on the FPGA . . . 73
4.3.3 ShiftShuffleNet on Embedded FPGA . . . 76
4.4 Recurrent Neural Network . . . 82
4.4.1 Long Short Term Memory . . . 82
4.4.2 Design and Training . . . 82
4.4.3 Acceleration on the embedded FPGA . . . 84

5 Conclusion and Future Work 89
5.1 Financial Option Pricing Algorithm . . . 89
5.2 Inline Application-Specific Caches . . . 90
5.3 Machine Learning Algorithm . . . 91

A Direct-Mapped Cache **93**
A.1 Code of the inline direct-mapped cache 93
A.2 Original and modified code of matrix multiplication 95

Bibliography **97**

List of Tables

1.1 Characteristics of the FPGA platforms . . . 4
1.2 Characteristics of the GPU platforms . . . 5
2.1 Value of the simulation parameters . . . 31
2.2 Performance on various platform . . . 31
2.3 Performance on various GPU platforms . . . 32
2.4 Resource utilization . . . 32
2.5 *Compute unit* vs *unroll* performance . . . 33
2.6 Resource utilization on AWS F1 . . . 33
2.7 Performance on AWS F1 . . . 34
2.8 Performance on PYNQ Z2 . . . 34
2.9 Resource utilization on PYNQ Z2 . . . 34
3.1 Performance and resource utilization for various implementations of matrix multiplication (16x16 matrices). . . . 48
3.2 Performance and resource utilization for various implementations of the Lucas-Kanade algorithm. . . . 53
3.3 Performance for various implementations of bitonic sorting applied to arrays with different sizes N, and using a cache line size of 64 bytes. $L_{\max}$, in bytes, is the maximum on-chip memory used (when limited). . . . 57
3.4 Performance and resource utilization of various optimizations on bitonic sorting applied to arrays with size $N = 2^{10}$. . . . 57
3.5 Effect of cache sizes on the performance of bitonic sorting . . . 58
4.1 Macro-structure of ShiftShuffleNet . . . 79
4.2 Quantization Result of the ShiftShuffleNet, "full" stands for single precision floating point number, "w[xx]" stands for xx-bit weights and "a[xx]" stands for xx-bit activation . . . 80
4.3 Performance comparison of the ShiftShuffleNet and previous works . . . 81
4.4 Parameters used to train the network . . . 84
4.5 Performance and resource utilization on XC7Z020 FPGA . . . 87

List of Figures

1.1 Moore's law in CPUs [52] . . . 3
1.2 Heterogeneous architectures . . . 3
1.3 Example of the roofline model [53] . . . 7
1.4 Example of the loop pipeline . . . 8
1.5 Example of the loop unroll . . . 9
1.6 Example of the dataflow model . . . 9
2.1 Design of the top-level structure . . . 21
2.2 Data-path of the MC method . . . 22
2.3 Dataflow optimization in cores connected by FIFOs . . . 22
2.4 Critical path in the step simulation . . . 23
2.5 Group path simulation . . . 23
2.6 Path simulations share expensive computation resources, such as square root operator . . . 25
2.7 Pipeline of group simulation . . . 25
2.8 Data-path of the PRNG algorithm . . . 26
2.9 Data-path of the new PRNG algorithm . . . 28
2.10 Static pipeline . . . 28
2.11 Multiple compute units . . . 29
2.12 Parallelism by dataflow . . . 29
2.13 Implementation of two parallel threads by 'Dataflow' . . . 30
2.14 Performance ratio between AWS F1 FPGA and Tesla P100 GPU in terms of execution time, power and energy consumption (log scale) . . . 34
2.15 Performance ratio between ZYNQ Z702 FPGA and GTX 950 GPU in terms of execution time, power and energy consumption (log scale) . . . 35
3.1 Design flow in SDAccel . . . 40
3.2 Inline cache . . . 41
3.3 Design flow with caches . . . 41
3.4 Diagram of a direct-mapped cache [54] . . . 43
3.5 Diagram of a 2-way set-associative cache . . . 44
3.6 Miss ratios for different numbers of lines, data sizes and line sizes for matrix A of matrix multiplication (log scale). . . . 49

3.7 Miss ratios for different numbers of lines, data sizes and line sizes for matrix B of matrix multiplication (log scale). 49
3.8 Miss ratios for different numbers of lines, data sizes and line sizes for the input image of Lucas-Kanade (log scale) 54
3.9 Miss ratios for different numbers of lines, data sizes and line sizes for bitonic sorting (log scale). 56
4.1 Artificial neural network example 62
4.2 Mathematical model of the artificial neuron 63
4.3 Mathematical model of the artificial neuron 64
4.4 Example of a sliding window for the convolution operation 65
4.5 Example of the max pooling [55] 67
4.6 Computation graph created by **TF** for a layer of the neural network . . 69
4.7 Computation graph auto-generated by **TF** for back-propagation 69
4.8 Design flow of the neural networks on an FPGA via the Tensorflow framework 70
4.9 Dimension of the sliding window and data accessing pattern 72
4.10 Sliding window generation 73
4.11 Diagram of the VGG16 net 74
4.12 Entire VGG16 on-chip 75
4.13 VGG16 partitioned and accelerated on multi-nodes. On AWS, F1 FPGAs can communicate only through the host DRAM control. 75
4.14 Partial layers on the FPGA to accelerate VGG16. 76
4.15 Small hardware architecture on an FPGA 76
4.16 Diagram of the depthwise convolution 77
4.17 Diagram of the ShuffleNet v2 block 78
4.18 Diagram of the shiftShuffle block 78
4.19 Diagram of the shift operation 79
4.20 Hardware architecture of the shiftShuffle block 80
4.21 Neural network designed for sensor data processing 83
4.22 Visual output of the neural network 84

Chapter 1

Introduction

1.1 Computing System

It has been over seven decades since the Turing machine model was invented by Alan Turing in order to represent algorithms for any computable function [37] to the ongoing competitions to build the exascale computing systems which perform $2^{60}(\approx 10^{18})$ FLOPS (i.e. floating point operations per second). Recent news tells that China, USA and Japan are competing to build such computing systems within 2020s. Tianhe-2, one of the fastest High-Performance Computing (HPC) system in the world relies on thousands of high performance CPUs and co-processors. Evidence shows that the exascale systems can consume about 1 GW power by extrapolating the results of the Tianhe-2 system[17]. To address these challenges, a novel architecture has to be designed in order to meet both the performance and energy requirements without consuming massive time to market [29].

1.1.1 Moore's Law

Moore's law describes the trend over time of the number of transistors in an integrated circuits. It was first found by Gordon Moore and was published in 1965. Figure 1.1 shows the number of CPU transistor against the dates. It clearly demonstrates that the number of transistors on the CPU doubles approximately every one to two years. However, this law almost reaches its end of validity nowadays due to the quantum effects dominating in the nanometer scale channel size and the difficulties to deal with the thermal issues (i.e. power wall). For the same reasons, the performance of single-core CPUs per area has reached its limit.

1.1.2 Heterogeneous Architecture

A homogeneous computing system, composed by a group of CPUs has been the preferred solution to build HPC systems for a while. However, it is no longer able to achieve a remarkable performance as demanded in modern data centers at exascale due to both the above-mentioned stop to the increase of single-CPU performance and power consumption [14]. The solution to this issue is provided by heterogeneous computing systems.

They are called heterogeneous because the co-processors are different from the host device, or they have different instruction set, or the programming languages and environments are different. The performance and energy efficiency can be achieved by the co-processors with specialized processing capabilities to handle particular tasks. The co-processors can be the Graphic Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs) and any other Application-Specific Integrated Circuits (ASICs). A heterogeneous architecture is illustrated in Figure 1.2. The co-processors communicate with the host processors through the Peripheral Component Interconnect Express

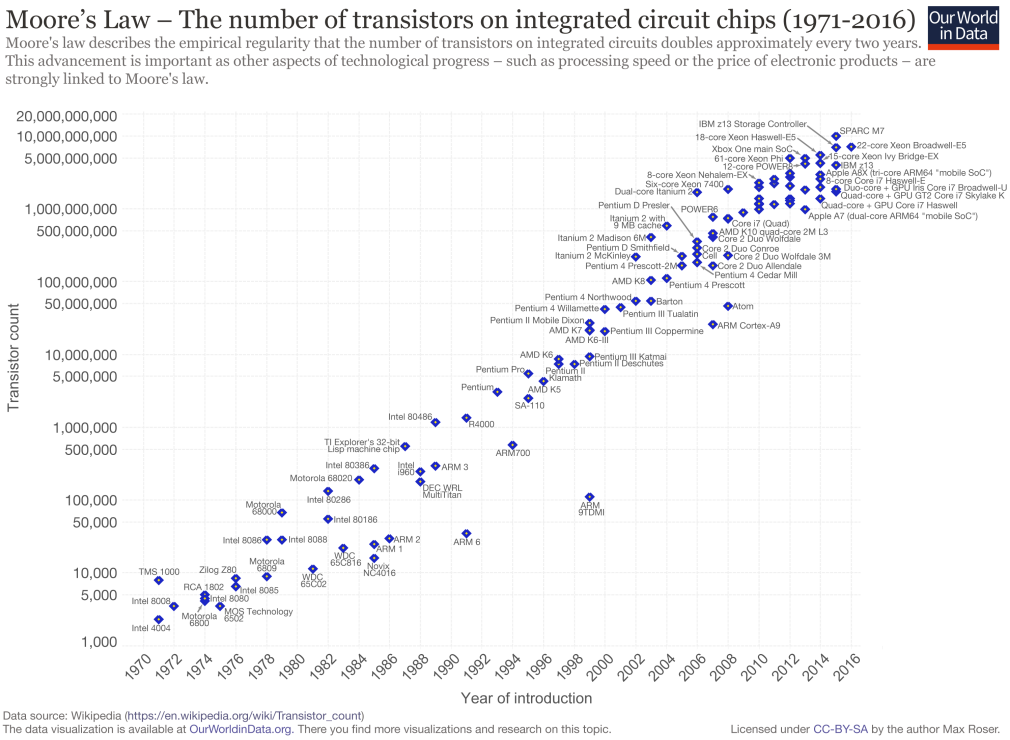

Figure 1.1: Moore's law in CPUs [52]

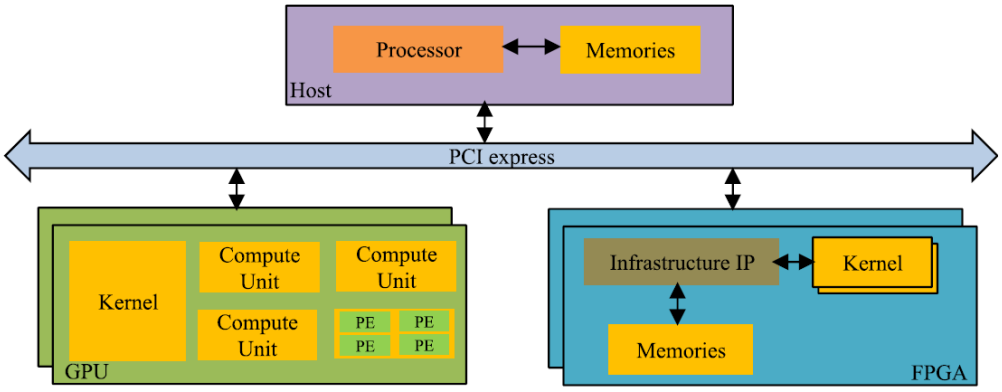

Figure 1.2: Heterogeneous architectures

(PCIe) bus. Each processor has independent memory. The co-processors may communicate each other via the same PCIe bus, or via other dedicated protocols (e.g. Aurora for Xilinx FPGAs).

Field-programmable gate array

An FPGA is a programmable integrated circuit which exploits a reconfigurable spatial computing architecture for a massive parallelism rather than the Instruction Set Architectures (ISAs). On modern FPGAs, such as the Stratix from Altera and the UltraScale families from Xilinx, there are up to millions of Configurable Logic Blocks (CLBs) and Flip-Flops, megabytes of on-chip the Block RAM (BRAMs), hundreds of multiply-and-accumulate units (DSPs), and many other dedicated hardware blocks, including ARM Cortex processors [61]. These CLBs can be connected via a hierarchy of reconfigurable interconnects (configurable wires) to perform complex combinational functions and sequential functions. The integration of the DSPs makes the modern FPGAs also eligible for floating point computing acceleration.

In this thesis, we targeted mostly two FPGA platforms:

1. Datacenter-calss Amazon Web service (AWS) F1 instance FPGAs, which are the 16 nm Xilinx UltraScale+ FPGAs with about 2.5 million logic elements and 6,800 digital signal Processors (DSPs).

2. Embedded-class: PYNQ (i.e. **PY**thon Productivity for Zy**NQ**) Z2 with a Xilinx Zynq SoC which integrates a 650MHz Dual-core ARM Cortex-A9 and a Zynq 7020 FPGA fabricated with 28nm technology and other peripherals such as the USB, SD card, HDMI, Ethernet and many other resources. The Z07020 FPGA has 13,300 CLBs, 630 KB BRAMs, , 512MB DDR3 DRAM and 220 DSPs on chip.

The characteristic of the two FPGA platforms are listed in Table 1.1. Compared to the datacenter-class FPGA, the on-chip resource of the ZYNQ Z0702 is approximately 24x less.

Table 1.1: Characteristics of the FPGA platforms

Device	Process[nm]	Frequency[MHz]	LUTs	FFs	BRAMs	DSPs
ZYNQ Z0702	28	100 to 200	53K	106K	280	220
UltraScale+ FPGA	16	250	1.2M	2.4M	4.3K	6.8K

Graphic processing unit

GPUs are specifically designed integrated circuits originally used to process graphical information such as images and videos. Currently they are widely used as the accelerators for parallel computations such as training machine learning algorithms. The architecture of the GPUs contains many computation cores also named Algorithm-Logic Units (ALUs) managed by a single control unit.

The GPU can be programmed in CUDA, a *proprietary* programming languages which provides a C/C++ syntax rules based language and programming environment, or the

very similar (but *open*) Open Computing Language (OpenCL) which is a framework to compile programs for executing on heterogeneous platforms. So the GPU can be easily used by the software developers.

However, it has been shown that these platforms are not very efficient with respect to the energy consumption [45] for many kinds of applications including the financial models and machine learning algorithms that we considered in this thesis. This issue has been addressed recently by using reconfigurable hardware platforms as accelerators.

The GPU platforms involved in this thesis are:

1. Nvidia GTX 950, chosen becuase it is available on a local server and it is also fabricated with the same 28nm technology as the Z0702 FPGA on the PYNQ-Z2

2. Nvidia Tesla P100 is an online service provided by Google Cloud Platform. The reason to choose the Nvidia Tesla P100 GPU is due to the fact that this GPU is fabricated with the same 16nm process, as the Amazon Web Service F1 FPGA. Thus we could fairly compare the performances on the two different accelerators.

The characteristic of the two platforms are listed in Table 1.2. Compared to the Nvidia GTX 950, the Tesla P100 GPU has 5x more parallel CUDA cores and 1.3x faster clock.

Table 1.2: Characteristics of the GPU platforms

Device	Process[nm]	CUDA cores	Frequency	Max. power[W]
GTX 950	28	640	0.9GHz	75
Tesla P100	16	3584	1.2GHz	250

Application-specific integrated circuit

An ASIC is an integrated circuit customized for a particular tasks rather than intended for general-purpose use. So the ASIC has limited programmability as an accelerator in a heterogeneous computing system. However, a well designed ASIC chip usually achieves the best performance and energy efficiency for the particular tasks running on it. For instance, the Google's Tensor Processing Unit (TPU) was particularly designed for machine learning inference models. It [28] was reported that the TPU server can achieve about 15x performance per Watt with respect to a decent GPU (K80) server.

1.1.3 High Level Synthesis

The Register Transfer Level (RTL) models are the predominant starting point for standard design flows for FPGAs and ASICs. These models are written in a Hardware Description Language (HDL), and then are synthesized, placed and routed by Electronic Design Automation (EDA) tools. However, this traditional design flow is losing stream. On one hand, it is very time-consuming due to the fact that any optimization to the

RTL model has to be painstakingly modified manually and extensively verified for the correctness. On the other hand, the standard software development flow, based on the principle of "write once, run anywhere" is attractive for hardware designers. Both Altera/Intel and Xilinx promise software-like development for applications that are entirely written in a high-level language and are then compiled and synthesized for heterogeneous CPU-FPGA platforms. This software-like design flow is named high-level synthesis (HLS). HLS design flow can dramatically reduced the design and verification costs, essentially eliminating the need to model the design at RTL.

HLS, on the other hand, promises the high performance and low power consumption of FPGA hardware, and the flexibility and re-targetability of software. In particular, parallel languages such as OpenCL that were originally developed to program GPUs can now be used to program the FPGA platforms.

Furthermore, HLS enables software-hardware co-design which aims at the synergistic design of hardware and software modules of a complex electronic system in order to achieve the best design point under constraints such as cost, performance, power consumption and time-to-market [48].

1.2 Performance Analysis

1.2.1 Amdahl's Law

Amdahl's law was presented by the computer scientist Gene Amdahl in the AFIPS Spring Joint Computer Conference in 1967 [4]. Amdahl's law is used to estimate the theoretical speedup in the latency of a system to process a fixed-size problem with respect to the resources involved. It is a formula as shown in (1.1), where p is the portion of the task that can be executed in parallel, S is the estimation of the speedup, N is an indicator of the resource used to accelerate the task, $r \geq 1$. In parallel computing, it can be used to evaluate the theoretical speedup by multiple processors when N represents the number of the processors. S is approximately inversely proportional to $1 - p$, the sequential portion of a task.

$$S(N) = \frac{1}{(1-p) + \frac{p}{N}} \tag{1.1}$$

1.2.2 Gustafson's law

When the problem size is no longer fixed, Amdahl's Law is no longer valid. Gustafson's law [22] named after John L. Gustafson and Edwin H. Barsis depicts another scheme for the parallel computation when the solvable problem sizes increase, as the computational power increases. Then multiple tasks can be run on N processors in parallel and achieve a speedup formulated by (1.2).

$$S(N) = (1-p) + pN \tag{1.2}$$

1.2.3 Roofline Model

The Roofline model is a very useful tool for performance estimation of a given application or compute kernel running on any processor [56] or accelerator. The roofline model illustrates the inherent hardware limitations such as the bandwidth, and potentially provides the developer with suggestions for further optimizations. Figure 1.3 is

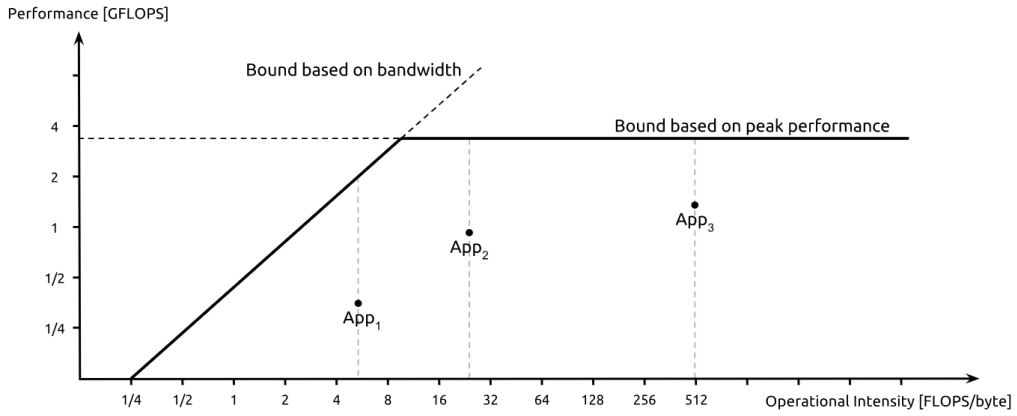

Figure 1.3: Example of the roofline model [53]

an example of the roofline model. In the model, the performance of an application is bounded by two factors: the peak performance of the processor/accelerator and the bandwidth of the memory. If the computation to communication ratio of an algorithm is high such as the App_3 in the Figure 1.3, the algorithm is called computation-bounded. In the other case when the computation to communication ratio of an algorithm is low such as the App_1 in the Figure 1.3, the algorithm is called memory-bounded. The optimizations are applied to the application according to whether it is a memory-intensive or a computation-intensive algorithm. In this thesis we will cover both compuattion-instensive algorithms (e.g. financial algorithms) and memory-intensive ones (e.g. Convolutional Neural Networks).

1.2.4 Optimization in HLS

Given an algorithm modeled in a high-level language such as C, C++ or OpenCL, several optimizations can be applied to improve its performance (and resource utilization) on an FPGA.

1. *Loop pipelining* starts new iterations of a loop before the completion of the previous ones. It is one of the best options for loop optimization in HLS, since it usually boosts the performance at a very low cost [15, p. 61]. The amount of clock cycles between the two successive loop iterations (inversely proportional to the throughput) is also called the "Initiation Interval (II)" of the pipeline (in the

best case, it can be one clock cycle). It is fully decoupled from the time it takes to complete one iteration, the "pipeline latency" or "pipeline depth". As shown in the Figure 1.4, each iteration in the loop has L operations. If the loop is not pipelined and assume each operation costs 1 clock cycle, the entire latency of the loop is NL clock cycles where N is the number of iterations in the loop. Once the loop is pipelined as the in the Figure 1.4, the entire latency reduces to (1.3), where D is the pipeline depth comparable to L.

$$\text{Latency} = II * (N - 1) + D \tag{1.3}$$

Usually, memory or data dependencies between successive iterations ("loop-carried dependencies") are the bottlenecks that increase the initiation interval. Several other synthesis techniques, e.g., array partitioning or loop interchange, can be applied to ameliorate this problem.

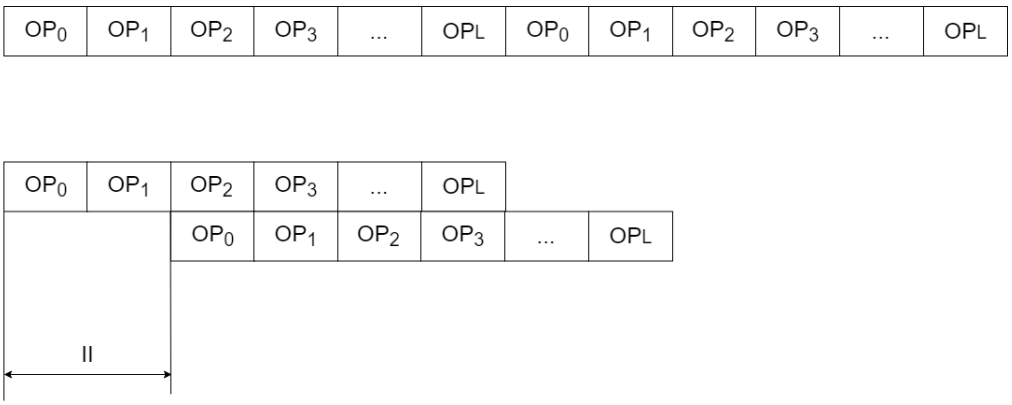

Figure 1.4: Example of the loop pipeline

2. *Loop unrolling* creates multiple copies of the loop body to be executed fully in parallel if their is no dependency among the iterations. In some cases it can achieve even more performance than by means of pipelining, but typically at a huge resource (i.e., area) cost. A loop can be fully or partially unrolled and in both cases the maximum performance can be achieved only by means of array partitioning and may require arithmetic evaluation restructuring (e.g., adder tree balancing) [15, p. 51]. In OpenCL (similar to CUDA), the loop over work groups (i.e. units of computation assigned to a GPU core) can be unrolled arbitrarily by definition. Taking the example in Figure 1.4, if the loop is unrolled by a factor of two, the performance will boost by a factor of two if there is no dependency as shown in Figure 1.5, where the loop is also pipelined.

3. *Compute unit* is another mechanism to increase parallelism that is similar to loop unrolling, but at a higher level. Languages like OpenCL in which the top-level

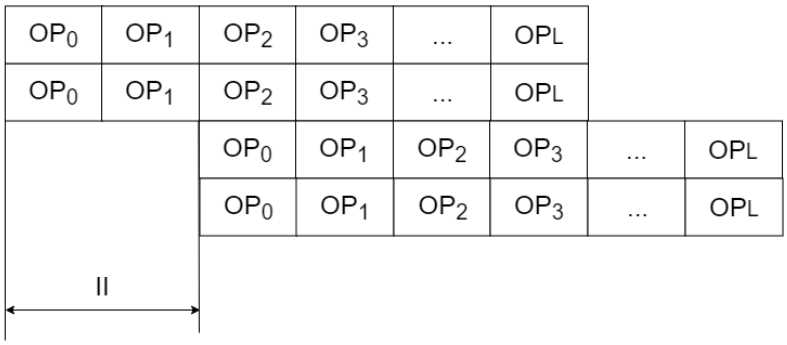

Figure 1.5: Example of the loop unroll

function (also known as "work item") is executed in a "doall" sort of parallelism allows one to instantiate multiple copies of a HW module implementing that function, and achieve a speedup that is only limited by available resources and memory bandwidth, similar to the multiple SIMD units on a GPU. Note that strictly speaking this multiple instantiation is only performed at place-and-route time, and does not affect HLS.

4. *DATAFLOW*. Computational processes in dataflow micro-architectures are controlled by the availability of the input data rather than a centralized finite state machine (FSM). Vivado HLS "dataflow" synthesis directive creates several independent processes connected by FIFO queues from functions in the original sequential C++ code. The developer can thus semi-explicitly drive parallelization from a high level, without the complexity of explicitly modeling threads in SystemC. As shown in Figure 1.6, each instances runs independently from other instances, in a task-level pipelining fashion, and is only driven by the input data from the FIFO. So this optimization explores task level parallelization, as opposed to loop-level fine-grained pipelining discussed above.. For each FIFO connecting two instances, $T^i_{in} = T^i_{out}$ where T^i_{in} is the average input data throughput into the i^{th} FIFO and T^i_{out} is the output data throughput to the $(i+1)^{th}$ instance. So the entire performance is limited to the "slowest" instance in the chain. In order to maximize the performance, we need carefully optimize each instance to match the throughputs.

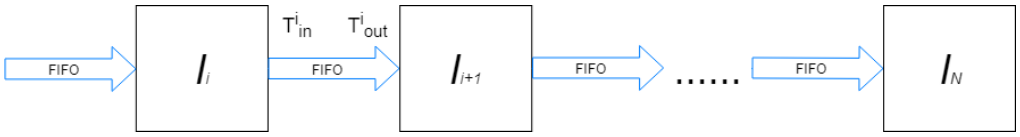

Figure 1.6: Example of the dataflow model

5. *Exploiting on-chip memory*. As discussed, the modern FPGAs such as the UltraScale, integrate thousands of independent BRAMs on chip for a total of many MBs of storage. The accesses to the BRAMs are both faster in terms of latency and more parallel than those to off-chip memories [58]. Most applications, especially the memory-bounded ones, achieve the best acceleration only by allocating frequently-accessed data that reside in off-chip memories into the on-chip BRAMs (or into LUTRAMs, which are not dedicated memories like BRAMs but rather a secondary usage of the small memories used to implement reconfigurable logic, i.e. LUTs). Each on-chip BRAM has two parallel ports which are configurable. The allocation of an array into the BRAMs has to be carefully managed in order to avoid causing bottlenecks for loop pipelining and unrolling. Memory partitioning or memory reshaping according to the user directives or based on automated analysis of access patterns of a given algorithm can dramatically increase the memory bandwidth and achieve a much higher level of concurrency. Unlike the memories on a GPU, where the maximum number of concurrent accesses to independent addresses is fixed by the GPU architect, on an FPGA it is configurable by the designers. However, more parallelism often implies a higher cost.

6. *Resource sharing* is very important in order to optimize the resource utilization and power consumption on the FPGA. Some resources are automatically shared by the the Vivado HLS tool. For instance, consider a loop partially unrolled and then pipelined. Some fast computation resources such as floating point multiplication unit can be shared by multiple threads (created by the loop unrolling) if the loop has a large initiation interval due to some slow resources such as the memory read or write. Vivado HLS also provides some synthesis directives that can help the designer to force the resource sharing of function cores or computing units that will consume too many resources.

7. *Optimizing kernel interfaces*. On a GPU, the global memory interface subsystem receives memory read or write requests from the threads or work items that are executing on its compute units, and *coalesces* these requests whenever possible, in order to match both the available memory word size and bus burst transfer capabilities. For example, 16 accesses to adjacent properly aligned 32-bit integer array elements can be grouped *automatically at runtime* into a single 512-bit memory read, or to a burst of 4 128-bit memory reads, depending on the DRAM interface width. This reduces both the number of accesses to the external memory and the total access time. On an FPGA, these groupings must be performed manually and at compile time by specifying the kernel interface data-width.

1.2.5 Power and Energy

Power and Energy on chip

The power consumed by a logic gate is composed by the dynamic power and the static power. The dynamic power is proportional to capacitance and the frequency. Thus the total power consumed by a device (e.g. FPGA) is $P = P_d + P_s$ the sum of the dynamic power P_d consumed by the active transistors and the total static power P_s of the device. P_d is given by (1.4), where C is the capacitance of transistors and interconnects that are switching, V is the device voltage and f is the operation frequency. The static power depends on the characteristic of the device and on an FPGA it can typically be treated as a constant value given a choice of FPGA, since voltage scaling and power gating are typically not used in these devices.

$$P_d = CV^2 f \tag{1.4}$$

The energy consumption by an application is given by (1.5), where t is the execution time of the application. In any case, the static energy consumption can be reduced only by reducing the execution time. For an application on an FPGA, we have $t = \frac{cc}{f}$, where the cc is the number of clock cycles to run the application. So the dynamic energy $E_d = CV^2 cc$ mainly depends on the active area and the number of clock cycles.

$$E = P_d t + P_s t \tag{1.5}$$

Energy for memory access

The energy consumed by each memory access is another important factor for accelerator design. [46] shows that the energy per access to the DRAM is about 200 time that for the local register files. The energy per access to the on-chip BRAM is much less than that to the DRAM. So reducing the number of accesses to off-chip DRAM not only reduces the execution time but also shrinks energy consumption.

1.3 Thesis Structure

The thesis is composed by five chapters, addressing the design and optimization of low power and high performance FPGA accelerators.

1. This chapter presented the high demands of low power and high performance computing systems and the heterogeneous hardware architectures designed to meet both the power and performance requirements.

2. The second chapter discusses the optimizations of computation-intensive algorithms. The application used is the Monte Carlo method applied to financial option pricing models.

3. The third chapter proposes a software-defined inline cache to accelerate the memory-intensive algorithms, such as the matrix multiplications (at the core of AI applications), image processing algorithms, and database algorithms.

4. The fourth chapter proposes various kinds of hardware architectures and optimizations to accelerate several machine learning models such as convolution neural networks using FPGAs.

5. The fifth chapter summarizes the achievements in this thesis and discusses the future work.

Chapter 2

Acceleration of Financial Algorithms

Everyone in the world is more or less involved in a very complex economy system. For instance, most people hold one or more financial assets as a portfolio such as the bank deposits, debits, stocks, directives and etc. The complexity of these products in the financial market grows faster than ever before. It makes the financial sector one of the hottest areas in the world. In addition, the growing number of complex financial derivatives boost the risks to both the participants and the financial system [30]. A mathematical model that can predict the market behaviors in time would be helpful for the investors to avoid reckless decisions.

2.1 Financial Options

A financial option is a type of contract between two parties in the derivative market. With this contract, the holders or the buyers have the right but not obligation to buy (i.e. call option) or to sell (i.e. put option) an instrument at a pre-settled price (i.e. strike price) at the specified dates or at any time in the valid period of the contract. Taking a call option as an example, if the stock price at the future date has a higher price than the strike price, the holder has the right to buy the instrument at the strike price and then sell it in the financial market to make a profit. Neither gains nor lose nothing the call option holder from the market if the stock price is lower than the strike price. Definitely, the right to make a profit without any possibility of loss is not a free product in the market. The buyers need to pay an amount of money which is also call premium for this right. The premium is decided by the two parties according to the instruments, the strike prices and also other assets in the market. As well, the holder of the options can also transfer this right to others at a certain price. Then, the option price is also a function of time. Since there is a lot of profits in this zero-sum game, many researchers and institutions focus on finding a good way to evaluate the option prices.

According to the expectations of the prices of the instruments in the markets, the speculators choose from the two type of options (i.e. call and put) to make a profit from it. According to the constrains set to exercise the contract with the strike price, the options are classified into different styles such as vanilla and exotic options.

2.1.1 European Vanilla Option

The European vanilla option is one of the common and simple options in the derivative markets. The contract of the European option can only be exercised at the specified date T which is also known as the option's maturity date preset in the contract.

For the call option holders, if the prices of the assets (noted as $S(t)$) at the future time T is higher than the strike price (noted as K), they can expect a profit of $S(T) - K$ in the market. Otherwise when $S(T) <= K$, they neither earn nothing from the market nor lose anything since they have no obligation to exercise the contract. The profile they can make is called payoff. So, for the European call option, the payoff is evaluated by (2.1).

Similarly, for the put option holders, the profit is made only when $S(T) < K$ as opposite to the call option. The payoff for the European put option is given by (2.2).

$$P_{Call} = max\{S(T) - K, 0\} \tag{2.1}$$

$$P_{put} = max\{K - S(T), 0\} \tag{2.2}$$

2.1.2 European Barrier Option

European barrier option has the similar rules as the European vanilla option with one more constrains. The contract can be only exercised at the maturity date T and the

asset prices has not to be over the barriers levels (e.g. upper barrier, lower barrier or both) set in the contract at anytime of the preset time period. For simplicity, we assume that the contract starts at the time $t = 0$ and ends at the time $t = T$.

$$P_{Call} = max\{S(T) - K,0\} \quad \forall t \in (0,T) \Rightarrow S_d \leq S(t) \leq S_u \tag{2.3}$$
$$P_{put} = max\{K - S(T),0\} \quad \forall t \in (0,T) \Rightarrow S_d \leq S(t) \leq S_u \tag{2.4}$$

where S_u and S_d are the upper bound and lower bound respectively.

2.1.3 American Option

Compared to the European vanilla option, American option gives the holders more flexibilities to exercise the contract. The exercise can happen at any date prior to the maturity date T. The holders definitely have to think about the strategies to find the best time to exercise the contract in order to maximize the payoff from the flexibility of the American options.

2.1.4 Asian Option

The Asian option which is also called average value option. Different from the vanilla options, the payoff price of the Asian option depends on the average price of the stock over the time period T. So this is an exotic option. According the way to calculate the average price, the Asian option is divided into two types. (2.5) and (2.6) define the first type and the payoff price is calculated by using the arithmetic mean. (2.7) and (2.8) define the call price and put price respectively by using the geometric mean.

$$P_{Call} = max\{\frac{1}{T}\int_0^T S(t)dt - K,0\} \tag{2.5}$$
$$P_{Put} = max\{K - \frac{1}{T}\int_0^T S(t)dt,0\} \tag{2.6}$$
$$P_{Call} = max\{e^{\frac{1}{T}\int_0^T ln(S(t))dt} - K,0\} \tag{2.7}$$
$$P_{Put} = max\{K - e^{\frac{1}{T}\int_0^T ln(S(t))dt},0\} \tag{2.8}$$

2.2 Option pricing models

2.2.1 Black Scholes Model

In option pricing modeling, one cannot miss the Black–Scholes model also noted as **B-S** model. The **B-S** model is a partial differential equation that describes the prices of the capital assets by a constant mean variance, also called volatility which is a statistical measure of the standard deviation of the returns [16]. Some assumptions such as

the frictionless market, absence of arbitrage and free transactions make the **B-S** model simple to analysis, especially to guarantee analytical solutions. One of the basic assumptions is that the price of a risky asset is subjected to the geometric Brownian motion, a stochastic differential equation (SDE) with constant volatility [5] shown in (2.9). And there is at least one riskless asset such as the saving account in the market with interest rate r shown in (2.10)

$$dS(t) = \mu\, S(t)dt + \sigma S(t)dW(t) \tag{2.9}$$

$$dP(t) = rP(t)dt \tag{2.10}$$

where $S(t)$ is the asset price also called stock price, $P(t)$ is the price of the risk-free asset, μ is the average growth rate of the stock price and it is equal to r, the interest rate of the risk-free instrument, due to the no-arbitrage assumption [26], σ is the constant volatility measured by the standard deviation of the stock prices and $W(t)$ is a stochastic process also known as Wiener process.

The analytical solution is obtained in (2.11) by Itô's lemma [26].

$$S(t+\Delta t) = S(t)e^{(r-\frac{1}{2}\sigma^2)\Delta t+\sigma\epsilon\sqrt{\Delta t}} \tag{2.11}$$

where $\epsilon \sim N(0,1)$, the standard normal distribution. From this analytical solution, one can derive the Black-Scholes equation [26] for the option pricing for the European options and American options.

The numerical solution (2.12) can be easily derived from 2.11 by the Euler discretization method [10] under the condition that $\Delta t \ll 1$. The numerical solution is extremely useful when deal with the exotic option problems.

$$S(t+\Delta t) = S(t)(1 + r\Delta t + \sigma\epsilon\sqrt{\Delta t}) \tag{2.12}$$

2.2.2 Heston Model

Instead of a constant variance in the **B-S** model, a time-dependent variance $\sigma(t)$ in (2.9) can make the model more accurate to predict the behaviors of the asset prices. A series of models had been designed in the last decades like the Leland's model [5] and Heston model. The Heston model which was designed in 1993 [24] is widely used in the option pricing evaluation. It describes the stock prices and the volatilities by a group of stochastic differential equations. (2.13) models the stock prices with the similar form in (2.9) and (2.14) models the instantaneous volatilities by a special stochastic process known as the Cox–Ingersoll–Ross (CIR) process [3].

$$dS(t) = rS(t)dt + \sqrt{V(t)}S(t)dz_0(t) \tag{2.13}$$

$$dV(t) = \kappa(\theta - V(t))dt + \sigma_v\sqrt{V(t)}dz_1(t) \tag{2.14}$$

Where $z_0(t)$ and $z_1(t)$ are two Wiener processes, ρ is the correlation factor between them, $Cov[dz_0(t), dz_1(t)] = \rho dt$ and $-1 < \rho < 1$. $\sigma(t) = \sqrt{V(t)}$ is the volatility of the

stock price, so that $V(t)$ must be positive. In (2.14), θ is the long-run mean variance, κ is the speed of mean reversion(the rate at which $V(t)$ reverts to θ) and σ_v is the volatility (Standard deviation) of $V(t)$.

For a short time $\Delta t \ll 1$, $V(t)$ can be assumed to be constant, so that Itô's Lemma can be applied to (2.13) as in the **B-S** model, which is then simplified as in (2.9). The numerical solution for (2.14) is also obtained by Euler discretization[10] with full truncation scheme avoiding negative values under the square root. The final solutions are shown in (2.15) and (2.16).

$$S(t+\Delta t) = S(t)(1 + r\Delta t + \sqrt{V(t)^+}(\rho\epsilon_1 + \sqrt{1-\rho^2}\epsilon_0)\sqrt{\Delta t}) \tag{2.15}$$

$$V(t+\Delta t) = V(t)^+ + \kappa(\theta - V(t)^+)\Delta t + \sigma_v\sqrt{V(t)^+}\epsilon_1\sqrt{\Delta t} \tag{2.16}$$

where $\epsilon_0, \epsilon_1 \sim N(0,1)$ and $V(t)^+ = max(V(t), 0)$.

2.3 Monte Carlo Method

The Monte Carlo method (MC method) named from the casino Monte Carlo in Monaco, is one of the most widely used approaches to simulate stochastic processes, like the behaviors of multiple electrons/holes in a semiconductor structure or a stock price modeled with Black-Scholes in this thesis. This is especially true for those physical and mathematical problems that rely on numerical solutions due to the lack of closed forms. In most of the solutions, the Monte Carlo method requires a large amount of random numbers according to certain distribution within certain range.

In this chapter, we adopted the Monte Carlo Method to simulate the stochastic process described by the **B-S** model and Heston model in order to estimate the expectations of the payoff prices of a given instrument and a given option. By this method, both the vanilla options and the exotic options can be analyzed in the same framework.

To do so, we need to have to define some terms first.

Path simulation (**PS**): Since the stock price is a stochastic process over time, we call the computations from the initial stock price to the final stock price at time T as a path simulation. We define the number of simulations as N_S. Usually N_S is a large number used to compute the expectations of the payoff of the given option in order to guarantee the results converging to the true expectation.

Step simulation (**SS**): To do the **PS** by the numerical solution, it's necessary to partition the time period of an given option. Given the time interval $(0, T)$, for convenience, we applied a uniformly partition into M steps and these steps were denoted as $t_0 = 0, t_1, t_2, \ldots, t_M = T$. In order to guarantee $\delta t = \frac{T}{M} \ll 1$, M should be a large value. We call the computation from $S(t_{k-1})$ to $S(t_k)$ as a step simulation. Each **SS** needs one or more normally distributed random numbers. This number is determined by the option pricing models.

Performance indicator (**PI**): For a given option problem and a given option pricing models, the computations for each **SS** is independent from the computation platforms and the values of N_S and M. So the mean time to do one **SS** or the average amount of **SS** per unit time is suitable for the performance indicator of a platform. In this thesis, we use the second definition as the performance indicator, which is the number of **SS** that the platform can compute per second as shown in (2.17).

$$\mathbf{PI} = \#\mathbf{SS}/s \tag{2.17}$$

2.3.1 Random Number Generator

As described, it requires one or more normally distributed random numbers for each **SS** in the MC method applied to the option pricing models So the total amount of random numbers is proportional to the $N_S M$. In general, any non-uniform probability distributed random generator can be realized by the uniform distribution random number generator and a function that converts the two distributions. The most common method is named inverse transform sampling.

Mersenne-Twister Algorithm

In order to achieve faster convergence and accurate results at the end of the MC method, a key aspect is the quality of the random numbers that it uses. In general, the software random number generation algorithms are Pseudo-Random Number Generators (PRNG) due to the fact that these algorithms depend on the initial seeds for initialization and the generated sequence of random numbers are periodic. For MC method, which consumes plenty of random numbers, Mersenne-Twister algorithm is the best choice for its long periodicity.

Box-Muller Algorithm

After obtaining the independent uniform distributed random numbers, one can get the normal distribution random numbers by the inverse transform sampling. As shown in (2.18) [18], it is the transformation function from a uniform probability variable $x \sim U(0,1)$ to Gaussian distribution variable $y \sim N(0,1)$.

$$y = T^{-1}(x) = \sqrt{2} erf^{-1}(2x - 1) \tag{2.18}$$

However, the accurate computation of inverse error function is very expensive. Most the algorithms [7] numerically compute the inverse error function by the piece-wise linear approximation [19] or the polynomials which is usually small and efficient but insufficient in resolutions [49]. Here, in this thesis, we adopt another method named Box-Muller transformation, which generates high quality normally distributed numbers but consumes a large computing resources.

So the procedure is to generate two independent random numbers U_0 and U_1 from the uniform distribution on the unit interval (0, 1) by **MT** algorithm.

$$Z_0 = \sqrt{-2 \ln U_1} \sin(2\pi U_0) \tag{2.19}$$

$$Z_1 = \sqrt{-2 \ln U_1} \cos(2\pi U_0) \tag{2.20}$$

Then by the (2.19) and (2.20), two standard normal distribution random numbers Z_0 and Z_1 are obtained.

2.4 Related work

The acceleration of the Monte Carlo method applied to the financial algorithms has been analyzed by many researchers. De Schryver, et al. [**de2011energy**] accelerated the Heston model applied to the European barrier option using both the 40 nm Nvidia Tesla c2050 GPUs and the 65 nm Virtex-5 FPGAs. In their experiments, even though the FPGAs achieved slower performance than the GPUs by about 4x, the power of the FPGAs was about 8x smaller than the GPUs. The average energy per step simulation of the FPGAs was about 2x better than the GPUs.

Tse et al. [**tse2010efficient**] implemented the Black-Scholes model applied to Asian options on a 65 nm Virtex-5 FPGA by manual RTL design and compared it with a 55 nm Nvidia Tesla C1060 GPU. In their report, the FPGA achieved 2.2x faster performance and about 6x less power than the GPU with a better technology node.

Inggs et al. [**inggs2014high**] designed their accelerators on various FPGAs using various high level synthesis tools including Altera SDK and Xilinx Vivado HLS, and proved that the HLS methodology was mature to achieve good results by comparing the results with multi-core CPUs and GPUs. In addition, they also discussed how to exploit the parallelism in the Monte Carlo method applied to financial algorithms.

2.5 Implementation and Architecture

The basic implementation is shown in Algorithm 1 and Algorithm 2. For the two algorithms, there are two nested loops over the N_S, the number of path simulations and M, the number of path simulation steps. We modified the Mersenne-Twist algorithm to generate two random uniformly distributed numbers in order to feed the Box-Muller algorithm, which takes two values as the inputs and then generates two random normally distributed numbers. In **B-S** model, the two random numbers are used for two simulation steps while in the Heston model, they are used for updating the stock price and the value of the volatility respectively. In order to reuse the code for different random number generation algorithms, option pricing models and options, in this paper, these algorithms are implemented mainly in six classes with interfaces to communicate and perform the simulations, shown in Figure 2.1.

Algorithm 1 Black-Scholes model

Input: parameters for the stock and option
Output: payoff price
Initialization: Random number generators
1: **for** $i = 1$ to N_S **do**
2: **for** $k = 1$ to M **do**
3: $U_0, U_1 \leftarrow MersenneTwist()$
4: $\epsilon_0, \epsilon_1 \leftarrow BoxMuller(U_0, U_1)$
5: $S_{t_{k+1}} \leftarrow Price(S_{t_k}, \epsilon_0)$
6: $S_{t_{k+2}} \leftarrow Price(S_{t_{k+1}}, \epsilon_1)$
7: $k+ = 2$
8: **end for**
9: $P_{option}[i] \leftarrow Option(S_t[], K)$
10: $i{+}{+}$
11: **end for**
12: **return** $P_{payoff} = mean(P_{option})$

Algorithm 2 Heston model

Input: parameters for the stock, volatility and option
Output: payoff price
Initialization: Random number generators
1: **for** $i = 1$ to N_S **do**
2: **for** $k = 1$ to M **do**
3: $U_0, U_1 \leftarrow MersenneTwist()$
4: $\epsilon_0, \epsilon_1 \leftarrow BoxMuller(U_0, U_1)$
5: $S_{t_{k+1}} \leftarrow Price(S_{t_k}, V_{t_k}, \epsilon_0)$
6: $V_{t_{k+1}} \leftarrow Volatility(V_{t_k}, \epsilon_1)$
7: $k{+}{+}$
8: **end for**
9: $P_{option}[i] \leftarrow Option(S_t[], K)$
10: $i{+}{+}$
11: **end for**
12: **return** $P_{payoff} = mean(P_{option})$

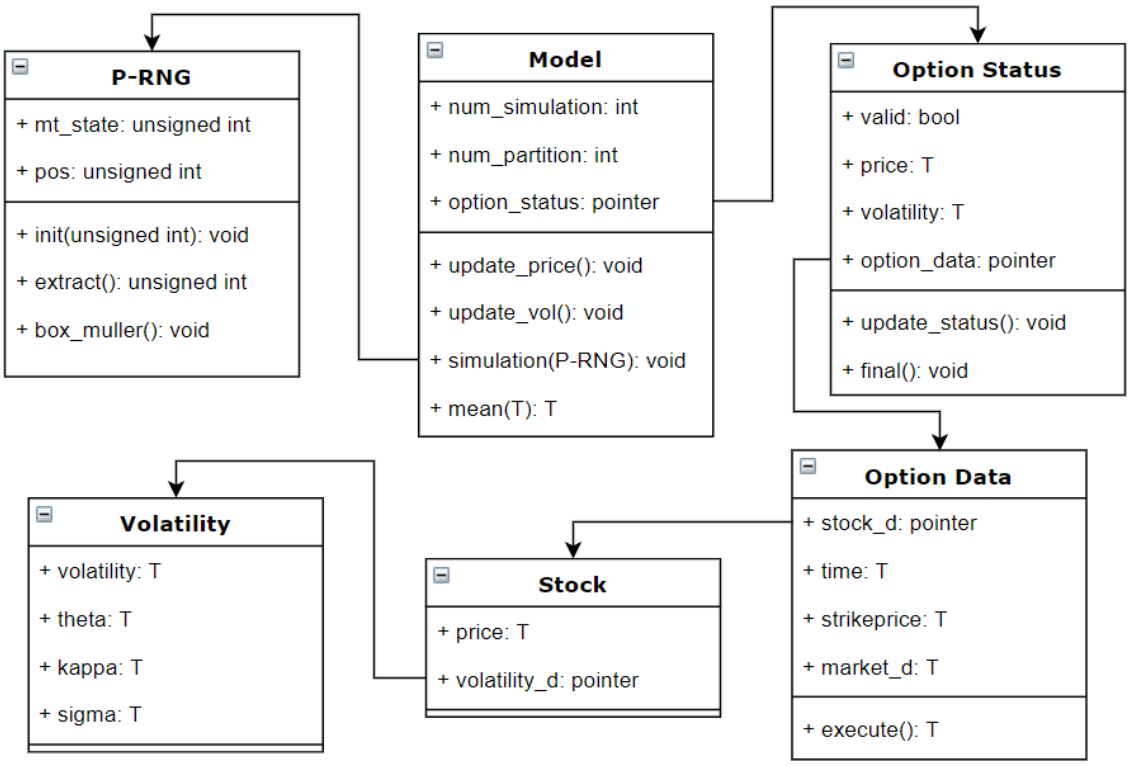

Figure 2.1: Design of the top-level structure

We use high level synthesis tools, like Xilinx Vivado HLS that can synthesize high-level C, C++, OpenCL and SystemC programs into register transfer level (RTL) netlists. We describe next a simple C++-based strategy that both simplifies model management and improves hardware design.

Both the algorithms 1 and 2 can be translated into micro-architecture on the FPGA as shown in Figure 2.2. At the initial phase, the controller reads from the memory the seed to initialize the PRNG and the parameters of the option. Then the simulation engine starts to update the stock price step by step over. Finally after N_S path simulations, the options' mean payoff is estimated and written back to the memory.

2.5.1 Optimization

Dataflow

The first optimization we applied is the "Dataflow" as introduced in the first chapter, it explores the task level parallelization and synthesizes the modular design efficiently. We applied this optimization and then got the architecture as shown in Figure 2.3.

In this architecture, if any connected modules have mismatched throughput, faster consumer or faster generator, one of the module has to stall since the FIFO size is limited and it reduces the overall efficiency. The target is to optimize each module and match

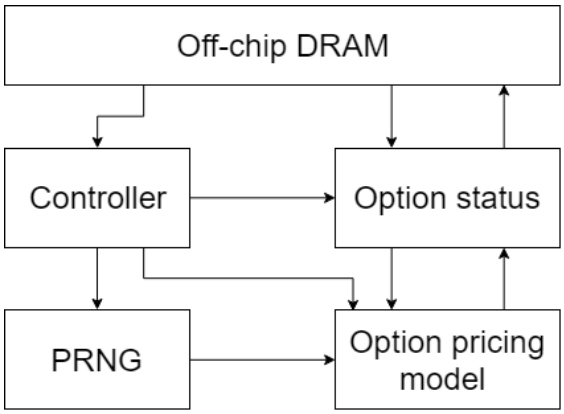

Figure 2.2: Data-path of the MC method

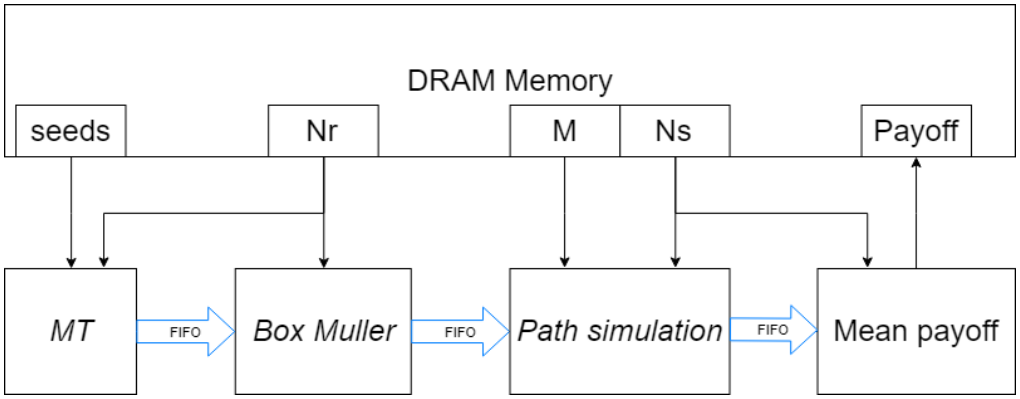

Figure 2.3: Dataflow optimization in cores connected by FIFOs

the throughput of each module.

Step simulation optimization

The first optimization applied here is to pipeline the inner most loop of the path simulation as shown in the two algorithms. However, the identity II is not able to be achieved due to the critical feedback loop in the data-path in the stock price updating phase, as shown in Figure 2.4 by the orange line. The slow floating-point multiplier creates a critical path in the design. The solution to this issue is to share the faster instance to multiple slow modules as a compromise. In addition, all the path simulations are independent from each other.

As shown in Figure 2.5, the PRNG continuously generates the independent normal distributed random numbers and distributes them to multiple path simulations by a MUX operated by the controller. In this design, the clock frequency can be higher since the floating-point multipliers has multiple clock cycles to finish the computation. At the end of the path simulations, the mean payoff price is estimated. The number of the

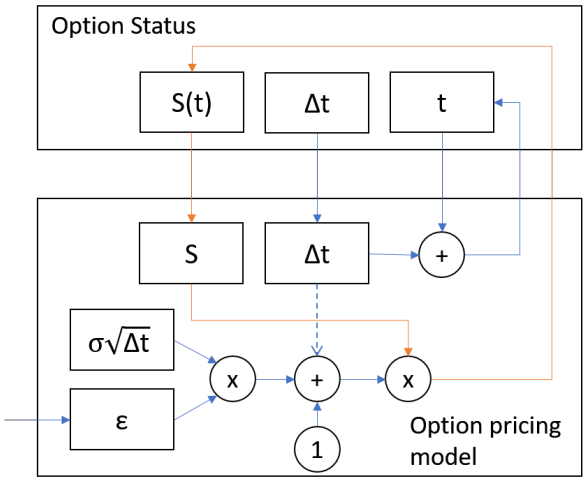

Figure 2.4: Critical path in the step simulation

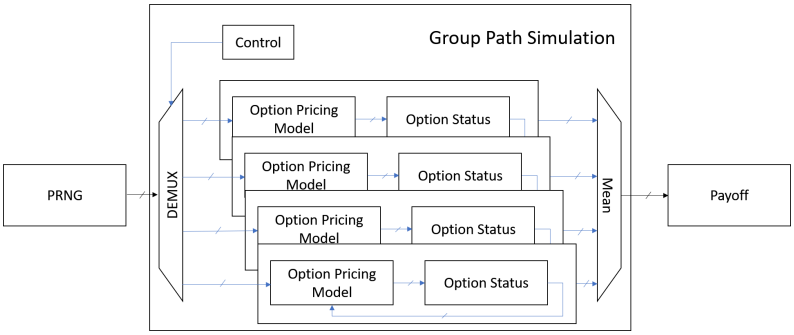

Figure 2.5: Group path simulation

multiplexed path simulations sharing one PRNG is denoted as N_G. The identity II is achievable by a proper value of N_G.

Note that while these simulations are independent, their implementations can share some expensive computational resources such as the *sqrt*, as shown in Figure 2.6. Hence, an additional level on top of Figure 2.4 uses a counter i to iterate over N_G and control the selection of random numbers and stock prices, where N_G is the number of path simulations sharing a PRNG.

Although the latter hardware micro-architecture is much more complex than the original one in Figure 2.4, by using HLS synthesis directives we minimize the changes of the software algorithm, as shown in Algorithm 3. Figure 2.7 demonstrates the pipeline of the modified algorithms with group simulations. In the original algorithm, the stock price estimation stalls the pipelining while the optimized algorithms has no longer the issue since the stock price in the successive iteration $S_{i+1}(t)$ is independent from the previous one $S_i(t)$.

Algorithm 3 Group path simulation

Input: parameters for the stock and option, id, N_S and M
Output: payoff price

Initialization:
prng(id)
opM(optionData)
for $i = 1$ to N_S/N_G **do**
  for $k = 1$ to M **do**
    for $g = 1$ to N_G **do**
      this loop is pipelined
      $\epsilon_0, \epsilon_1 \leftarrow$ prng.generate()
      opM.OptionStatusUpdate(option[i, g], ϵ_0, ϵ_1)
    end for
  end for
end for
return $P_p$ayoff = mean(option[].price())

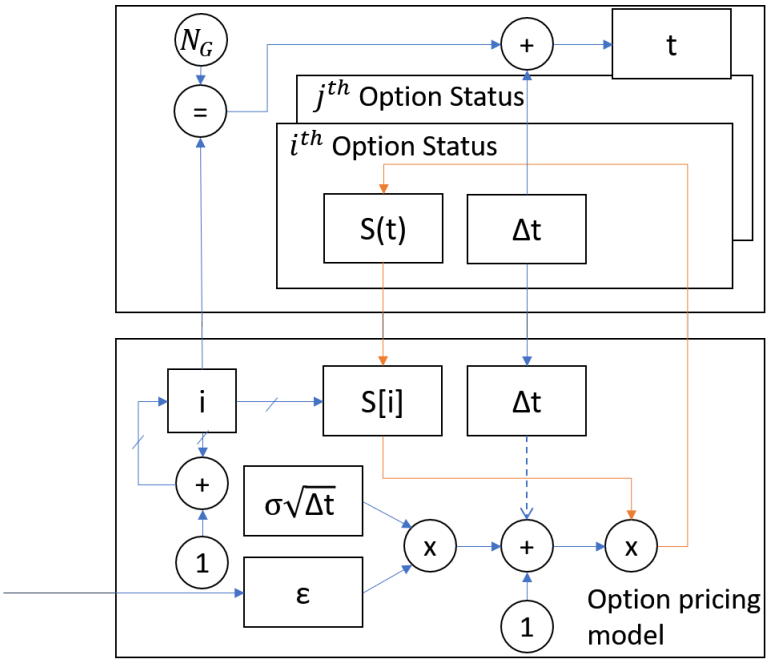

Figure 2.6: Path simulations share expensive computation resources, such as square root operator

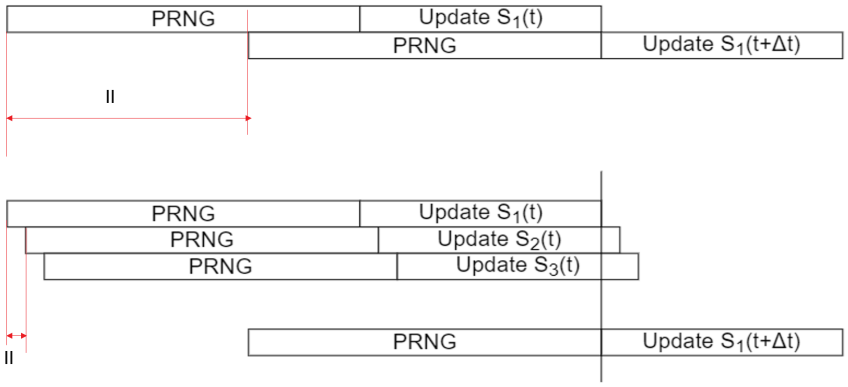

Figure 2.7: Pipeline of group simulation

PRNG

In the original **MT** algorithm, there are 624 states which is initialized by the seed and then updated during the random number generation. The state memory is allocated to block-RAM (BRAM), which has at maximum two ports on the modern FPGAs. To generate one random number it requires four distributed states, so there are three reads and one write from the BRAM, as shown in Figure 2.8. It means that the one random

number generation costs two clock cycles. Considering that the pipelined Box Muller algorithm is able to generate two random numbers in every clock cycle with two random numbers as the inputs. In order to match the throughputs of the two modules, we need to speedup the **MT** module by 4x. Since the random numbers are out-off order, the simplest way is to allocate multiple independent **MT** modules to achieve the desired performance. However, this method is not able to exploit the parallelism in the **MT** algorithm. The more efficient way is to partition the state memory into multiple parts to increase the memory access capability. Each independent BRAM can holds up to 18kb data (18-bit interface), if the array allocated to the BRAM is small, the unused storage is wasted. In this thesis, in order to maximize the efficiency of the resource utilization, we partitioned the state memory into two parts (even and odd index) as shown in the Figure 2.9. In addition, we duplicated the **MT** to achieve in total 4x performance.

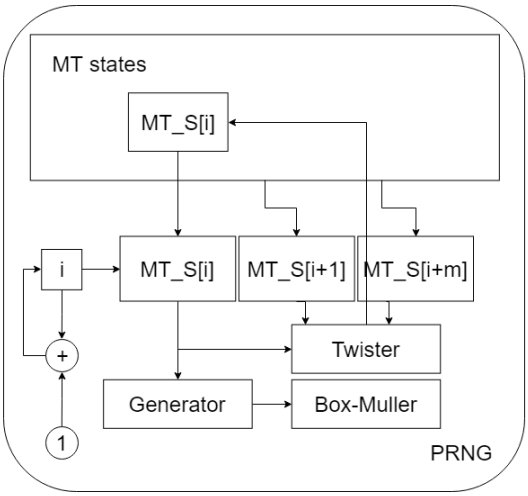

Figure 2.8: Data-path of the PRNG algorithm

Path simulation optimization

Since all the path simulations are independent, it is possible to deploy multiple instances of the micro-architecture shown in Figure 2.5 on the FPGA to maximize the overall throughput. The number of instances on the FPGAs are denoted as N_c. In this thesis, we provides three approaches to realize this implementation on the FPGAs.

Unroll

In the first approach, we aimed at exploring the parallelization by loop unrolling. In the software design, the N_c copies of the basic modules are programmed to run in parallel in the inner-most loop, which is unrolled by the HLS tools via directives. The pseudo-codes are shown in Algorithm 4 Figure 2.5. The arrays in the pseudo-codes have

Algorithm 4 Unroll algorithm

1: **for** $i = 1$ to $N_S/N_G/N_c$ **do**
2: **for** $k = 1$ to M **do**
3: **for** $g = 1$ to N_G **do**
4: **for** $j = 1$ to N_c **do**
5: $\epsilon_0, \epsilon_1 \leftarrow PRNG[j]$
6: $S_{t_{k+1}}[g][j] \leftarrow Price(S_{t_k}[j], V_{t_k}[g][j], \epsilon_0)$
7: $V_{t_{k+1}}[g][j] \leftarrow Volatility(V_{t_k}[g][j], \epsilon_1)$
8: **end for**
9: **end for**
10: **end for**
11: **end for**

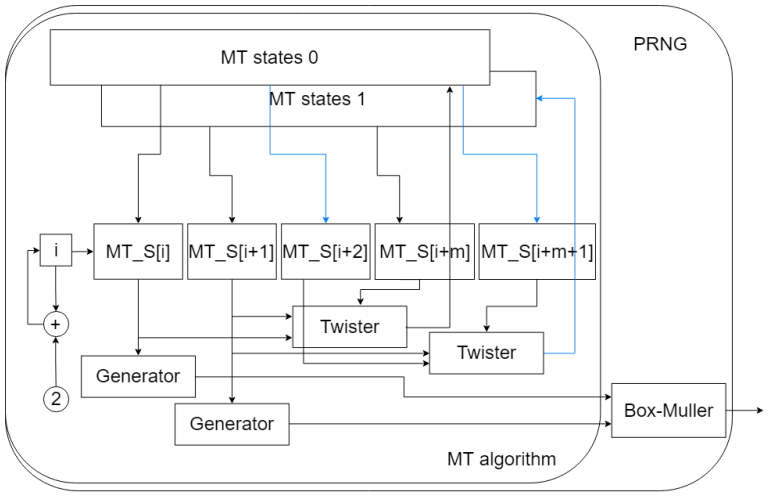

Figure 2.9: Data-path of the new PRNG algorithm

to be properly partitioned. In this case, all the concurrent threads are controlled by the same FSM of the step-simulation and the path-simulation as shown in Figure 2.10. One benefit of this architecture is that all the threads can communicate with each other and share the resources on chip. The drawback is that the developers have to change the source codes and the synthesized kernel is very large.

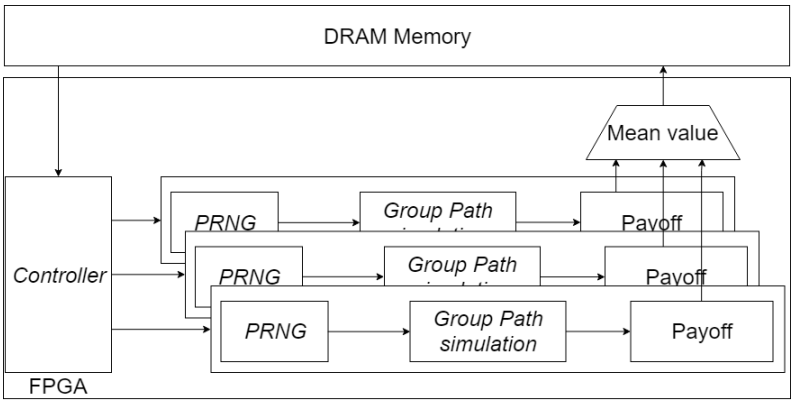

Figure 2.10: Static pipeline

Compute Units

In the second approach, we instantiate the N_c instances on the FPGA as multiple compute units during the "fabric linking" phase of the SDAccel, instead of the "fabric compilation" phase. It can be done easily by passing the number of compute units to the linker as an option in the 'Makefile'. The HLS tool synthesizes a smaller (less heavily unrolled) version of the architecture shown in Figure 2.10 as one independent compute

unit. The N_c instances work as completely independent threads, can even implement different option models, and communicate directly with the host CPU, as shown in Figure 2.11.

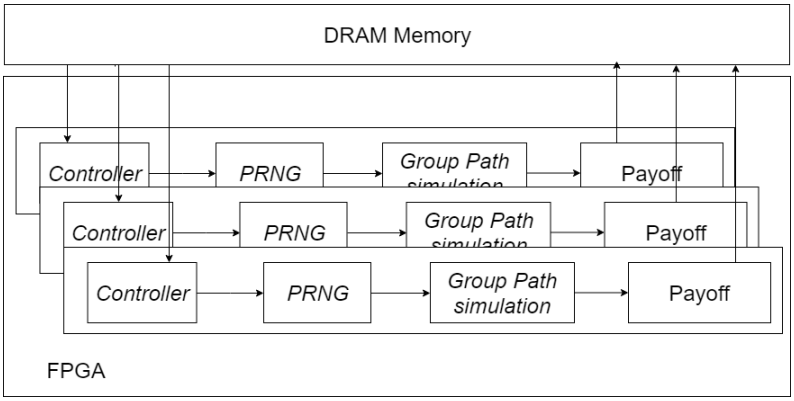

Figure 2.11: Multiple compute units

Dataflow

Since the 'DATAFLOW' directive in the VIVADO HLS explores the task level parallelism as mentioned above. We group the modules in Figure 2.3 as a single task. With the directive, the HLS tool automatically generates multiple identical tasks running in parallel in a single kernel. As shown in Figure 2.12, this architecture combines the features of the two architectures in the Figure 2.10 and the Figure 2.11. All the independent threads are initialized by the same inputs and the number of simulations are divided by them. The payoff price are estimated by the average value from each thread and then is written to the outputs ports. The codes to implement the this parallelism are shown

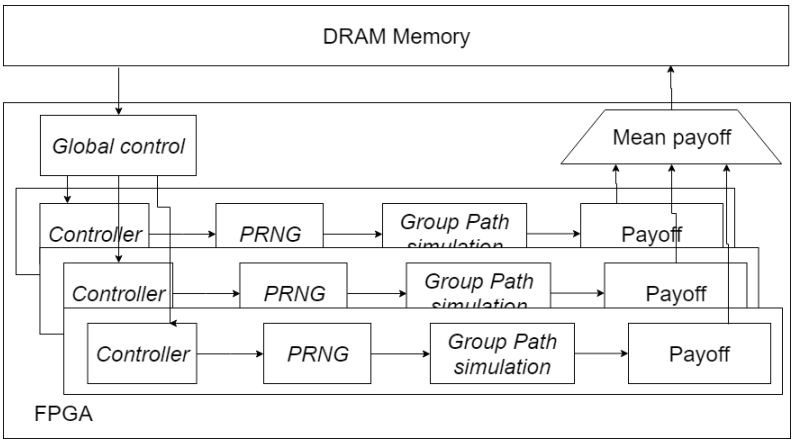

Figure 2.12: Parallelism by dataflow

in the Figure 2.13. In this implementation, there are in total two threads running in parallel.

```
  float call0, put0;
  heston<NUM_SIMS,EuropeanOptionStatus<float>, float> hs0(sd,vol, steps);
  float call1, put1;
  heston<NUM_SIMS,EuropeanOptionStatus<float>, float> hs1(sd,vol, steps);
  {
#pragma HLS FUNCTION_EXTRACT
#pragma HLS DATAFLOW
    launchSimulation(call0, put0, g_id, hs0, each_sims*steps<<1, each_sims);
    launchSimulation(call1, put1, g_id<<1, hs1, each_sims*steps<<1, each_sims);

  }
```

Figure 2.13: Implementation of two parallel threads by 'Dataflow'

2.5.2 Performance Indicator

The performance indicator is defined by (2.17) in the previous section. We can give the formulation of this parameter under the optimizations we applied. If we can obtain the performance of the hardware architecture we have in Figure 2.3, the overall performance then is given by a multiplication of N_c. Since N_c is determined by the FPGA resource, we need also analyze the hardware utilization of such architecture.

As analyzed above, the pipelined "Box Muller" block generates on average two Gaussian distribution random numbers per clock cycle. It means that the hardware architecture can process 2 **SS**s for **B-S** model or 1 **SS** for Heston model in each clock cycle. Then we obtained (2.21) and (2.22), where f is the device clock frequency. Of course, the two formulas give the upper bound on performance.

$$\mathbf{PI_{B-S}} = 2f \times N_c \tag{2.21}$$

$$\mathbf{PI_{Heston}} = f \times N_c \tag{2.22}$$

2.6 Results

For the sake of comparison, we use the value of parameters listed in Table 2.1 in all implementations, where $N_G = 32$ for **B-S** model and 64 for Heston model. The reason to choose these values is to guarantee an execution time on the CPU not beyond tens of seconds. On the accelerators such as GPUs and FPGAs, in order to measure the time more accurately, we computed the average running time of many repeated executions with large amount of simulations, and then re-scaled the results to perform the same amount of computations as the Table 2.1.

Table 2.1: Value of the simulation parameters

Name	N_c	N_s	N_G	M
Value	8	65536	32/64	1024

2.6.1 Running on CPUs

Since the algorithms are coded in C++, the code is compiled by **g++** with '**-O3**' optimization level. The performance of each option on the AWS F1 CPU (i.e. a Intel(R) Xeon(R) CPU E5-2686 v4 @2.30GHz core) is shown in Table 2.2. The CPU time for the Heston model of the European barrier option is about 4x less than that for the European vanilla option, because simulation can be stopped once the stock price goes beyond one of the barriers (in about 44236 paths out of 65536).

Table 2.2: Performance on various platform

Model	Option	F1 CPU [s]
B-S	European vanilla	3.56
B-S	Asian	3.88
Heston	European vanilla	5.16
Heston	European barrier	1.25

2.6.2 Acceleration by GPUs

We also tested the algorithms on the Nvidia GTX 950 GPU on local server and the Nvidia Tesla P100 GPU platform on Google cloud platform, by rewriting the models in **CUDA** to achieve the best performance on the GPUs. We report the actual power consumption from the Nvidia System Management Interface rather than the maximum power of each GPU. Table 2.3 lists the execution time, power and energy consumption of the four tests on various options. The results shows the Tesla P100 GPU is about 1600x faster than the F1 CPU for **B-S** models on the two options and about 1200x faster for the Heston models on European option and 300x faster on the European barrier option (since the SIMD model cannot take advantage of early termination).

2.6.3 AWS F1 FPGA

Comparison between various optimizations

On AWS, we first compared two optimizations, one acceleration by unrolling the inner most loop and the other one acceleration by suing multiple compute units. In order to have a fair comparison, we need to control the resource utilization, the number

Table 2.3: Performance on various GPU platforms

Model	Option	GTX 950 [ms]	Power [W]	Energy [J]	Tesla P100 [ms]	Power [W]	Energy [J]
B-S	European	11.15	84	0.937	2.4	170	0.408
B-S	Asian	11.17	84	0.938	2.11	170	0.359
Heston	European	26.3	91	2.39	4.31	181	0.78
Heston	European Barrier	26.13	87	2.27	4.33	180	0.779

of the PRNG modules, and the number of total computations of the two implementations. The resource utilization in the Table 2.4 is the approximate value for the two implementations.

Table 2.4: Resource utilization

Model	Option	LUT %	LUTMem %	REG %	BRAM %	DSP %
B-S	European	10	1	6	3	9
B-S	Asian	11	1	7	4	10
Heston	European	11	1	9	3	11
Heston	Euro. barrier	11	1	8	3	11

By the (2.21) and (2.22), we could estimate the lower-bound execution time by the total **SS** dividing **PI** for the test parameters listed in Table 2.1. The frequency of the F1 FPGA is 250MHz, so that the execution time is 16.78ms for **B-S** and 33.55ms for Heston model.

Table 2.5 lists the execution times of the two implementations. The performance of the independent compute unit implementation is close to the estimated lower-bound execution time while the inner loop unroll is about 1.5x slower.

Full Device Implementation

In order to compare the performance and energy profile with the GPU, we designed a large implementation on the F1 FPGA. We obtained the actual power consumption from the AWS F1 power reports provided by the platform.

From Table 2.7, we can conclude that the AWS F1 FPGAs have comparable performance to a Tesla P100 GPU, with a significantly (4x) lower energy consumption. These comparisons are also shown in the Figure 2.14.

2.6.4 Embedded Platform: ZYNQ 7020

Apart from the AWS F1 FPGA, we also tested our design on the embedded platform PYNQ Z2. By the same method, we could estimate the lower bound of the execution

Table 2.5: *Compute unit* vs *unroll* performance

Model	Option	F1 FPGA by UNROLL	F1 FPGA by CUs	Speedup CU/UNROLL
B-S	European	28.76ms	17.07ms	1.68x
B-S	Asian	25.92ms	17.1ms	1.52x
Heston	European	47.5ms	33.84ms	1.4x
Heston	Euro. barrier	46.8ms	33.82ms	1.38x

Table 2.6: Resource utilization on AWS F1

Model	Option	LUT %	LUTMem %	REG %	BRAM %	DSP %
B-S	European	66	7	38	23	71
B-S	Asian	70	8	42	31	80
Heston	European	62	8	37	20	77
Heston	Euro. barrier	63	9	39	20	77

time, we got 0.112s for **B-S** models and 0.224s for Heston models under 7ns clock period and $N_c = 2$. Note that this platform is not realistic for the implementation of financial applications, which are executed in large data centers, but we would still like to report it as a reference.

The algorithm execution times that we obtained on the PYNQ Z2 platform are listed in Table 2.8 and resource utilization are listed in Table 2.9. The execution time is very close to the estimation. The power information was obtained from the VIVADO power estimator, which is the best reference for the embedded platform, which unlike the AWS F1 does not have any mechanism for reporting the power consumption at runtime.

Of course, the performance of the Z7020 FPGA is slower than the F1 FPGA because it is much smaller (and cheaper). However, the embedded FPGA still has 23x better performance than the F1 CPU and 1.7x less energy consumption than the P100 GPU and about 5x less than the GTX950 GPU for the same amount of computation.

These comparison are also shown in the Figure 2.15.

Table 2.7: Performance on AWS F1

Model	Option	Performance [ms]	Power [W]	Energy[J]
B-S	European	2.84	41	0.116
B-S	Asian	2.81	42	0.118
Heston	European	7.2	41	0.295
Heston	Euro. barrier	6.4	40	0.256

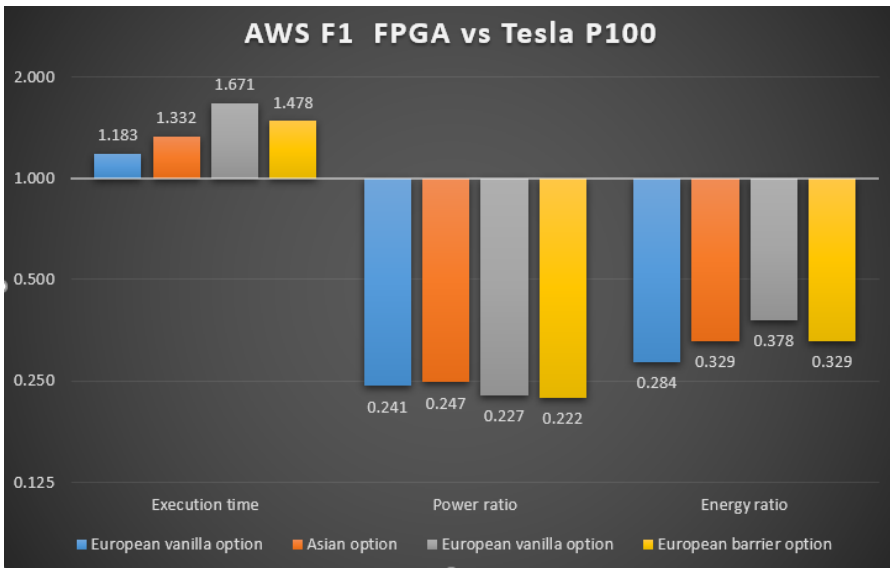

Figure 2.14: Performance ratio between AWS F1 FPGA and Tesla P100 GPU in terms of execution time, power and energy consumption (log scale)

Table 2.8: Performance on PYNQ Z2

Model	Option	Performance [s]	Power [W]	Energy[mJ]
B-S	European	0.118	2.02	0.24
B-S	Asian	0.117	1.84	0.21
Heston	European	0.225	2.1	0.47
Heston	Euro. barrier	0.236	2.06	0.48

Table 2.9: Resource utilization on PYNQ Z2

Model	Option	LUT %	LUTMem %	REG %	BRAM %	DSP %
B-S	European	45	11	35	10	70
B-S	Asian	47	11	36	13	68
Heston	European	49	15	40	10	75
Heston	Euro. barrier	52	14	38	10	88

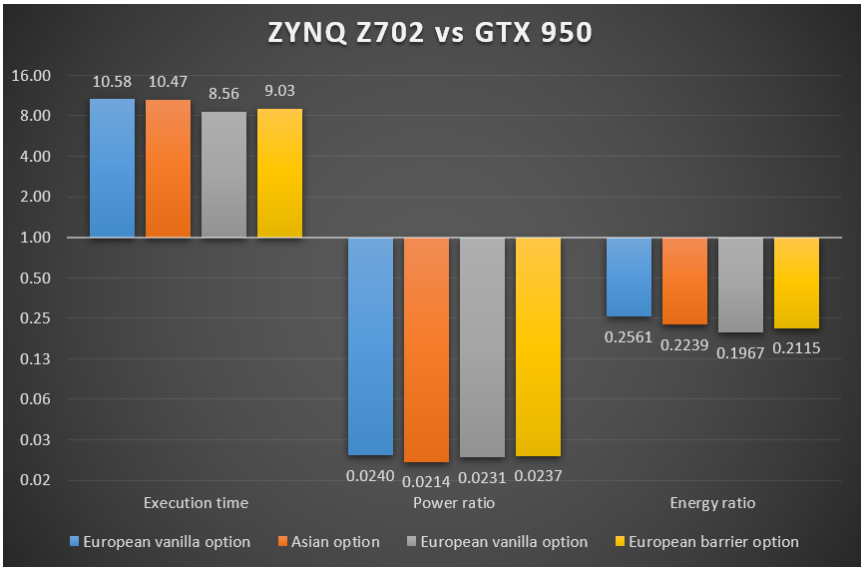

Figure 2.15: Performance ratio between ZYNQ Z702 FPGA and GTX 950 GPU in terms of execution time, power and energy consumption (log scale)

Chapter 3

Cache-Based Acceleration for Memory Intensive Algorithms

In many algorithms such as image classification and video processing, a large amount of data has to be transferred to and processed on the heterogeneous accelerators. In the roofline model, since these algorithms have a low computation to communication ratio, their performance is constrained by the interface bandwidth. For most memory-bounded algorithms, the data access pattern is not random and/or many of the data are accessed multiple times. For the first feature, developers can pre-fetch the data on chip according to the data access pattern in order to reduce the data conflicts and exploit the burst data access on the interface. For the second feature, local buffers (registers or block RAMs) can be used for data reusing in order to reduce the number of accesses to the external DRAM.

We designed an easy-to-use cache to explore these two features (spatial and temporal locality) of an algorithm automatically and help the designers to accelerate their algorithms on the FPGAs.

3.1 Background

3.1.1 Memory-intensive algorithms

Even though the HLS tool automates most the low-level labor-intensive transformations from high-level code to RTL, many decisions must still be made by the developers, and extensive code rewriting, especially for the memory-intensive algorithms, is sometimes needed in order to obtain the best performance on an FPGA.

In the previous chapter, we presented the ways to accelerate a computation-intensive algorithm mainly by the synthesis directives such as the loop pipeline and memory partitioning. Since the latency of memory-intensive applications is particularly significant in FPGAs due to off-chip memory bandwidth limitations, code rewriting and exploiting data reuse with on-chip BRAMs are inevitable in order to ameliorate both the performance and the energy consumption.

From the roofline model shown in Figure 1.3, the first application has very low computation to communication ratio and its best performance is not able to reach the fully potential of the device. The reuse of data can significantly enlarge the computation to communication ratio and then enable one to fully exploit all the computation resources on chip.

3.1.2 Cache Related Work

Modern CPUs generally include up to three levels of cache in order to reduce both data access time and energy. As the level increases, both latency and cache size (hence access power and energy) increase. These caches implement different access, replacement and coherency strategies to achieve the best *average performance* for all kinds of algorithms. Research on improving general-purpose caches is abundant.

Many researchers addressed the acceleration of memory-intensive algorithms on FPGAs by exploiting the highly configurable on-chip memory architecture. For example, Cheng et al. [11] developed a trace analysis method to detect relations among all memory accesses. Performance was greatly improved by caching independent data in separate local memories. Adler et al. [2] used BRAMs as statically-managed scratchpads rather than dynamically-managed caches, and described a management system for different levels of local storage. Choi et al. [12] implemented a multi-ported cache based on the so-called live-value table, aimed at a system architecture where both the host processor and multiple accelerators are on the same chip. In their approach, both the processor and the accelerators access the same off-chip memory via a single custom multi-port cache, which of course may become a performance bottleneck. Putnam et al. [39] provided a cache-based solution to simultaneously increase performance and reduce power consumption, since external DRAM accesses require much higher power than on-chip SRAM. In this design methodology, the CHiMPS HLS tool first compiles the high-level code (written in C) to an intermediate representation and then the caches

are optimized according to the memory access patterns. Similarly, Winterstein [57] also used the LLVM intermediate language to maximize the utilization of BRAMs to accelerate a specific algorithm (tree reflection).

Our approach is inspired by some of these works, in particular to reduce access conflicts by *using a separate cache, possibly with a different architecture, for each source code array* mapped to external DRAM.

3.1.3 Motivations

The motivation to design the cache aims at avoiding all the code rewritings that optimize the access to large arrays of data. Even though the designed caches do not exclude the high level verification, our approach is able to avoid the significant *verification cost* of these changes, because caches are guaranteed to always deliver the right data. In the context of algorithms like those targeted by this research (which have regular access patterns), they can even "guarantee" good performance, where the guarantee is as good as the test cases which are used to select the cache parameters and to verify the performance post-synthesis.

In past work, the caches are usually designed as concurrent HW modules. While this strategy offers some advantages, such as a better decoupling between the external memory and the IPs, it also has a significant disadvantage: it requires one to *change the accelerated kernel code* to access the caches via dedicated interfaces rather than directly access the source code arrays. This is incompatible with the strategy of providing a software-like design environment for FPGA hardware,

This issue motivated us to design the *inline caches*. In order to define the scope of this research on accelerating memory-intensive algorithms, we list some requirements for the caches:

1. Enabling significant performance acceleration with respect to code that was not written specifically for FPGA, and sometimes not even optimized for a GPU.

2. Improving execution energy consumption by targeting an FPGA platform, by reducing off-chip memory accesses, and by decreasing the execution time.

3. Providing the developers with the high level performance information about for the specific application and an easy-to-use source-level mechanism to configure the high level parameters such as the size of the cache

4. Supporting optimized use of external DRAM interfaces (e.g. DDR3 or DDR4) via advanced on-chip buses (e.g. AXI).

5. Enabling the use of HLS tools and preserving the standard HLS-based verification flow.

6. Requiring almost no changes to the original algorithms.

7. Not hampering the standard set of optimizations introduced in Chapter 1, and that are offered by the HLS tools.

3.2 High Level Cache Design

3.2.1 Hardware Design Flow

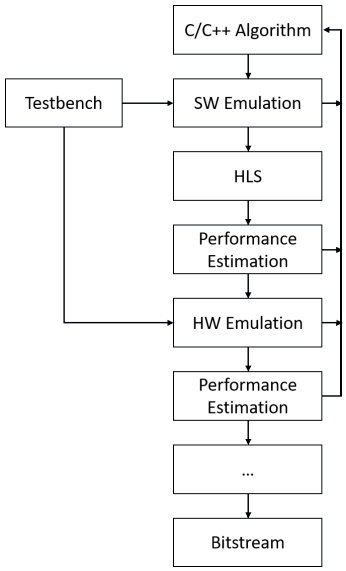

Figure 3.1: Design flow in SDAccel

As shown in Figure 3.1, the general design flow starts from the software (SW) emulation (i.e. C verification), which guarantees the functional correctness of the algorithm using properly designed testbenches. Both the algorithms and testbenches can be modeled in C, C++ or OpenCL and then are synthesized (i.e. "fabric compilation" in SDAccel) via Vivado HLS tools with the estimations of the resource utilization and the performance. According to the information in the estimation reports, the developers can further direct HLS towards the desired solution by the optimizations at the high level. During the following hardware (HW) emulation (RTL simulation) phase, SDAccel calls Vivado to connect (i.e. "fabric linking") the synthesized kernel IPs with other infrastructure blocks as shown in Figure 1.2 and launches a co-simulation between the RTL and the high-level testbench. This phase usually last longer time than the SW emulation, but it also generates a more accurate report of the performance, which in particular includes the latency due to the off-chip DRAM accesses.

As shown in Figure 3.2, the caches are directly "inlined" in the algorithms to be accelerated. In this way, *the "golden" code that has been functionally verified by SW emulation does not need to be changed for high-performance implementation.* Only the top-level module interface (which is typically much smaller and simpler than its often intricate

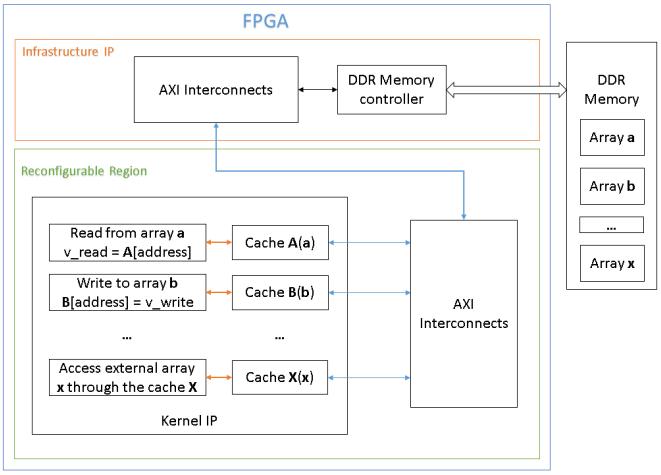

Figure 3.2: Inline cache

algorithmic code) requires some small changes, as illustrated below. In the resulting RTL, the caches are directly synthesized as part of the kernel IP.

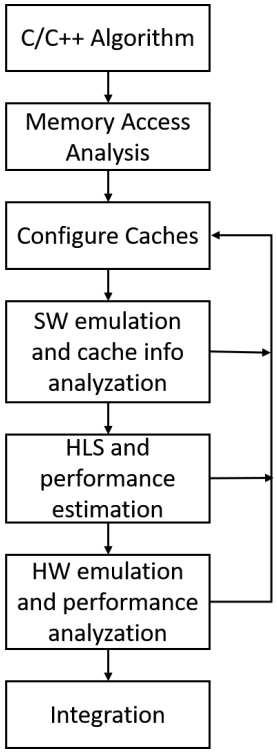

Figure 3.3: Design flow with caches

The new design flow is shown in Figure 3.3. Before applying the caches, the analysis of the external memory access traces is necessary to find the best cache configurations

to maximize the reuse of the data and minimize the miss ratio with an acceptable area cost. It requires the designers to make a few modification to the top-level function interface by replacing the original data types of the global array variables with a template cache data type. Then the following SW emulation can be used to verify the correctness of the modification, analyze the external memory accesses and to evaluate the performance of the cache by the hit ratio, which is automatically captured and printed by our cache models. Then the HLS, followed by HW emulation or actual FPGA prototyping to obtain more detailed external memory performance information, which can potentially lead to further optimizations.

3.2.2 Inline Cache Types

In this work, we propose and describe several kinds of inline caches, e.g., direct-mapped and set associative, selected based on the memory-trace pattern of the applications to be optimized. For each array mapped to global memory, a stand alone cache is implemented. It means that *performance is largely independent of the global memory addresses at which each array is allocated*, and that there are no conflicts between different arrays. Since accelerated kernels typically make fairly regular accesses to each array, this means that *real-time performance of our dedicated caches is much more predictable than that of traditional shared caches*, and can be comparable to that of manually managed scratchpads.

Direct-Mapped Cache

The direct-mapped cache is the most basic and intuitive cache. As its name indicates, every element in the external memory is mapped to a fixed position in the cache, according to the bit field of the address. Figure 3.4 illustrates a 256-byte direct-mapped cache caching the data in a 16K-byte (i.e. 14-bit address) memory. The address are partitioned to three fields according to the cache size and the block size. The bits fields in the middle of the address determine the line to be mapped in the cache, while the word bits define the offset within the cache line. The tag bits are used to examine whether a given address is cached in the corresponding line of the cache (“cache hit”) or not (“cache miss”). In the latter case, the cache fetches the correct data from external memory and updates the corresponding cache line and tag.

Even though each line contains multiple data, it is updated by a single AXI bus access, possibly using a burst (depending on the line and data bus bit widths). The write policy for the caches in our research is write-back, i.e., only the cache is updated with the new data initially, while the external memory is updated only when the cache needs to be flushed, either due to a read/write miss or due to the completion of the accelerator execution. As mentioned above, a standalone cache for each array in the external memory, which avoids all kinds of coherency issues for our caches.

In some algorithms, and in particular in the most massively parallel cases written

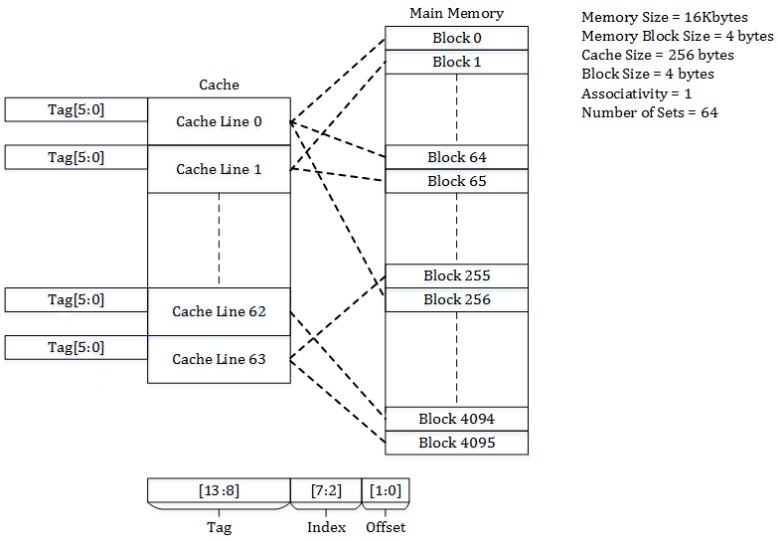

Figure 3.4: Diagram of a direct-mapped cache [54]

in languages such as OpenCL, the uses of each array argument of a kernel are either read-only or write-only. Hence, we designed a special cache for these read-only and write-only memory accesses in order to speed up the synthesis, reduce the cost, and improve the performance. As usual, we keep valid and dirty bits for each cache line, to indicate if it contains valid data from memory or data that needs to be written back to memory.

Set-Associative Cache

In some algorithms (e.g., sorting, FFT), *the successive data read to the external memory are not located at contiguous addresses*. In the worst case, the memory accesses with stride as same as the line size would cause the worst performance, since all accesses might miss the target in the cache. The set-associative cache is the a good solution for these algorithms.

Figure 3.5 illustrates an example of a 2-way set-associative cache, which has N sets and M words in each cache line. The data fetched from main memory can be stored in any line in a cache set. The replace policy in our example code is the Least Recent Used (LRU), but other algorithms can be implemented as well. In Figure 3.5, the *LRU* field records the request order of last access of each cache line. In this research, we use as time stamp (i.e., *LRU* value) the *request* counter, which was also used for statistical purposes.

Designers should carefully choose the number of ways and the replace policy of a set-associative cache when optimizing the performance, because a large number of ways causes higher resource utilization and the replace policy may form a very long critical path in the design.

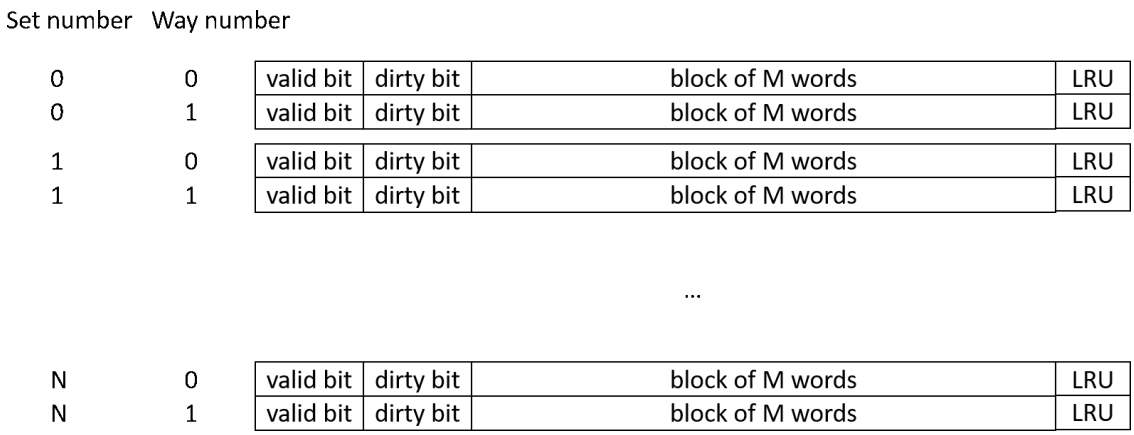

Figure 3.5: Diagram of a 2-way set-associative cache

Just like in the case of direct mapped caches, also for set-associative caches we have three variants: read-only, write-only and read-write configurations.

In this work we did not consider fully associative caches due to the high cost of the Content Addressable Memory.

3.2.3 Inline Cache Implementation

The inline cache in our work was designed in C++ by using a template class as shown in Listing A.1. The template arguments include the definition of the variable type of the element, the line size, the way size, the word size, the cache type and other parameters. The cache uses a multi-dimensional array to store the data fetched from the external memory and register-files to hold the tags, the valid bit and the dirty bit. The constructor of the class requires the base address of the corresponding off-chip memory array (typically the value of a pointer argument of the OpenCL kernel or C++ top-level function) as a mandatory argument to initialize the corresponding member variable which is used to compute the external memory addresses of each memory access. In HLS, the constructor is typically executed as part of the reset sequence of the HW block. The constructor and destructor of the inline cache take care of all the bookkeeping, from initializing the cache as empty (resetting all valid and dirty bits), to flushing an output cache and printing the statistics in a simulation context, when the accelerator completes its operation.

In the C++ or OpenCL algorithmic code to be implemented via HLS, the external memory is usually accessed by the `operator[]` or the `operator*` on a pointer passed from the interface. Hence, we overloaded the `operator[]` for the cache type, for uses

on both the left hand side (write) and the right hand side (read) of an assignment[1]. This allows us to change only the interface of the function to be synthesized, not its original source code, thus dramatically reducing the design time and the likelihood of coding errors. For instance, we show the modification from the original code of the matrix multiplication algorithm in Listing A.2.

Note that since the cache access functions (for reading and writing) are inlined into the high level kernel code, the synthesized kernel takes care of both executing the computation using the cached data, and reading/writing data from/to the main memory in case of misses. As we mentioned above, this somewhat reduces the achievable performance, but it dramatically simplifies the design flow and is consistent with OpenCL philosophy, where the work items themselves take care of moving the data from global to local memory. In future work we are planning to experiment with the use of separate processes to handle the caches.

In order to achieve the best performance, the data width of the AXI interfaces that are used to transfer a line to and from external DRAM should have the same size as a cache line, so that a read or write can be completed in one clock cycle (plus global memory latency in case of reads, of course). The kernel ip interface are automatically defined once the configuration of the cache is done. If the line length is larger than the global memory read size, then burst accesses will automatically be used by our design. This is one of the key advantages that the designer gets for free by using our caches.

Algorithm 5 and Algorithm 6 demonstrate how a direct-mapped cache reads or writes an address of global memory. The set-associative cache has the similar algorithms with a search function to find the correct line from each set and replace function to locate the line to be replaced once the cache misses.

The pair of variables *request* and *hit* are used as performance counters to enable cache parameter tuning also when an FPGA is used as a rapid prototyping platform, and can be accessed via FPGA-provided debugging mechanisms (e.g., via JTAG). The *valid* and *dirty* arrays have Boolean elements. The *tags* array contain unsigned integers of the appropriate length. The *array* array is used to store all the lines of data in the cache.

The two algorithms share a similar structure. Lines 1-4 handle cache hits. The address is split into three pieces, namely *tag*, *line* and *word*, then the value (or values, for the set-associative case) stored in *tags* is compared with the *tag* part of the address. If it is a hit, the following operation is the read from (or write to) *array*, on line 16. In both cases, the actual location of the data within the line depends on the value of *word*. If it is not a hit, then a new read from the external memory is necessary (after writing back the dirty line in case of a write or read/write cache). For special cases, a read-only cache does not need to check if a line is dirty and has only Algorithm 5 implemented.

[1]We managed to overload differently the read and write accesses to call a different cache access function, by exploiting an inner class as an agent [63].

Algorithm 5 Read data from direct-mapped cache

Require: 32-bit *addr* **and** *Cache* with a pointer *ptr_mem* to external memory
Ensure: $data = Cache[addr]$

```
1:  tag, line, word ← addr
2:  request ← request + 1
3:  if tag = Cache.tags[line] and Cache.valid[line] then
4:      hit ← hit + 1
5:  else
6:      if Cache.dirty[line] then
7:          location ← Cache.tags[line], line
8:          ptr_mem[location] ← Cache.array[line]
9:          Cache.dirty[line] ← false
10:     end if
11:     loc ← addr » LINE_BITS
12:     Cache.array[line] ← ptr_mem[loc]
13: end if
14: Cache.tags[line] ← tag
15: Cache.valid[line] ← true
16: return data ← Cache.array[line].slice(word)
```

Algorithm 6 Write data to direct-mapped cache

Require: 32-bit *addr* **and** *data* **and** *Cache* with a pointer *ptr_mem* to external memory
Ensure: $Cache[addr] = data$

```
1:  tag, line, word ← addr
2:  request ← request + 1
3:  if tag = Cache.tags[line] and Cache.valid[line] then
4:      hit ← hit + 1
5:  else
6:      if Cache.dirty[line] then
7:          location ← Cache.tags[line], line
8:          ptr_mem[location] ← Cache.array[line]
9:      end if
10:     loc ← addr » LINE_BITS
11:     Cache.array[line] ← ptr_mem[loc]
12: end if
13: Cache.tags[line] ← tag
14: Cache.valid[line] ← true
15: Cache.dirty[line] ← true
16: Cache.array[line].slice(word) ← data
```

The C++ code is listed in Listing A.1.

Cache-dependent HLS Optimizations

Applying the inline caches described in this work to the applications, developers usually obtain better performance compared to original algorithms. In order to further improve the performance, the algorithm- and cache-dependent optimizations are necessary. As mentioned above, the designed inline caches are compatible with HLS optimization directives. Application-specific post-optimizations include but not limited to the pipelining or unrolling a loop, and providing memory dependency directives. For example, a memory dependency is assumed to exist, if there is an array that is both read and written. Often, HLS may not be able to detect automatically if this memory dependency is true or not, and directives are required to optimize memory accesses by using knowledge coming from the programmer.

A very useful optimization provided in our design can be used when some array accesses in the code will be always hits (e.g., the access to array element $i + 1$ after accessing element i, if i is even and the line size is at least 2). When the address analysis performed by the HLS synthesis tool is not powerful enough to detect this situation due to complex address computations, this can be done manually or automatically (in the future work) by using two dedicate designed member functions of the cache class. The methods *retrieve()* and *modify()* can be used instead of the convenient operator "[ ]" to directly read or write respectively an element of the array by assuming that it is already in cache. These functions can dramatically improve the throughput by reducing the initiation interval of pipelined inner loops, like a convolution operation, which accesses the same array multiple times in the innermost loops. As shown in Figure 3.3, a SW or HW emulation must follow every modification to the source code by using retrieve() and modify(), in order to guarantee its correctness.

3.3 Applications

In this section, we presents how we applied the inline cache to optimize an application or an algorithm. The three algorithms [2] are matrix multiplication, Lucas Kanade algorithm and bitonic sorting. For each algorithm, we had three groups of implementations:

1. Implementation *without* memory optimization. It is called "External memory" implementation in this thesis.

2. Optimization by loading all data from external memory to on-chip BRAMs and keeping other optimizations using the same directives as the first group. It is

[2] In our previous work [35], we have reported more detailed results for many more applications.

called "on-chip memory" implementation in this thesis.

3. Optimization by a well configured and optimized inline cache and keeping other optimizations using the same directives as the first group. It is called "cache" implementation in this thesis.

We compared the performance and the energy consumption [3] of the three group of implementations in order to present the usefulness of the inline cache for helping accelerating memory intensive algorithms.

3.3.1 Matrix Multiplication

Matrix multiplication is the most basic algorithm in many scenarios such as the machine learning models. As well, it is also a typical memory-intensive algorithm. Here we test the multiplication of two matrices $A \in \mathbb{N}^{R\times R}$ and $B \in \mathbb{N}^{R\times R}$. The output matrix is $C \in \mathbb{N}^{R\times R}$, with the same size with A and B. Here we adopted integers as the data type for the matrices in order to reduce the latency caused by the floating point operations as presented in Chapter 2. We used two direct-mapped read-only caches for input matrices A and B, and a direct-mapped write-only cache for output matrix C.

Note that due to a limitation of the Vivado HLS tool, except the cache interface, we had to slightly modify the code of the "caches" implementations, in order to make the loop nest perfect – we incorporated the output matrix assignment into the last iteration of the innermost loop. This manual code change almost doubles the overall performance. Table 3.1 lists the performance, device power and resource utilization of all three groups

Table 3.1: Performance and resource utilization for various implementations of matrix multiplication (16x16 matrices).

Implementation	**Ext. Mem.**	**On-chip Mem.**	**Cache**	
Loop flatten	Yes	Yes	No	Yes
Exec. time (ms)	0.241	**0.027**	0.058	**0.031**
Power (W)	**0.507**	**0.471**	1.345	1.201
Energy (mJ)	0.122	**0.013**	0.078	**0.037**
BRAM	3	2	38	31
DSP	3	3	3	3
LUT	1792	1462	6588	5699
FF	3051	2237	16186	17794

of the implementations for 16x16 matrices. The caches for matrices A and C contained

[3] Due to the limitations of the Vivado power estimator, we estimated the power of the FPGA without the power consumption of the external DRAM.

one 16-word line each (i.e., one row of the matrix). The cache for matrix B contained 16 16-word lines, which was also the size of matrix B.

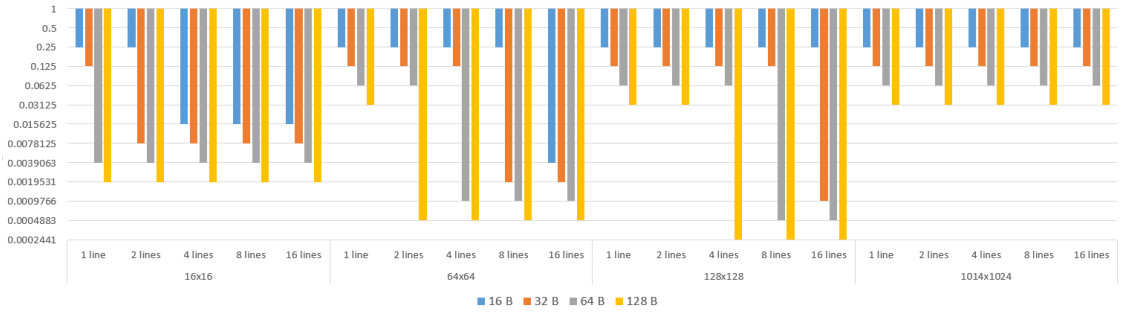

Figure 3.6: Miss ratios for different numbers of lines, data sizes and line sizes for matrix A of matrix multiplication (log scale).

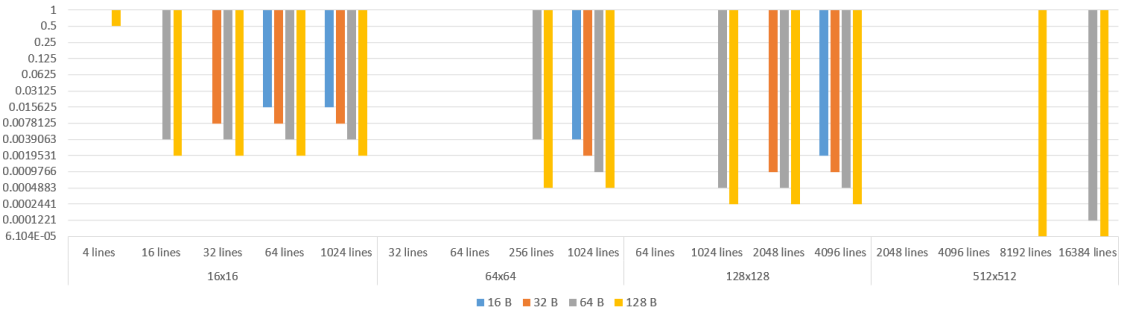

Figure 3.7: Miss ratios for different numbers of lines, data sizes and line sizes for matrix B of matrix multiplication (log scale).

Note how the cache-based implementation with loop flattening achieves essentially the same performance as the "ideal" implementation, where all data fits in the on-chip memory. Of course, the caches have a significant resource cost, which becomes particularly noticeable for computationally-simple algorithms like matrix multiplication. Moreover, the energy consumption of the best cache implementation is only 30% of that of the external memory implementation. This is without considering the energy consumed by the external memory itself, which would make the cache-based implementations even more efficient, due to the low miss ratio.

More complete results, for a broad range of matrix and numbers of lines, are reported in Figure 3.6 and Figure 3.7. As shown in Figure 3.6, the hit ratio of the caches applied to matrix A is highly dependent on the line size and is not affected by the number of lines in the cache until the cache can hold all the data in the matrix, when the miss ratio can be reduced to 0.02%.

The caches applied to matrix B have a different behavior, as discussed above. The miss ratio can be small only when the cache size is the same the matrix size as shown

in Figure 3.7, thus making caches useful for matrix B only in order to automatically perform burst accesses to global memory.

3.3.2 Lucas-Kanade Algorithm

The Lucas-Kanade algorithm uses a differential method to implement the optical flow function in the computer vision domain and it was developed by Bruce D. Lucas and Takeo Kanade [33]. It has been used to solve the feature tracking problem where two images taken close in time are analyzed to find small (thanks to time proximity) pixel displacements due to movements of various objects. J.Y. Bouguet [9] introduced a method to compute the optical flow velocities (e.g. moving objects in a video) based on the partial derivatives of images as shown in (3.1) and (3.2), (3.3), (3.4) and (3.5). For instance, this method can be applied to the videos taken by the traffic cameras on the road to surveil the status of the cars and the trucks.

$$I_x(x,y) = \frac{\partial Im(x,y)}{\partial x} = \frac{Im(x+1,y) - Im(x-1,y)}{2}, \tag{3.1}$$

$$I_y(x,y) = \frac{\partial Im(x,y)}{\partial y} = \frac{Im(x,y+1) - Im(x,y-1)}{2}, \tag{3.2}$$

$$G \doteq \Sigma_{x=p_x-w_x}^{p_x+w_x} \Sigma_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}, \tag{3.3}$$

$$b \doteq \Sigma_{x=p_x-w_x}^{p_x+w_x} \Sigma_{y=p_y-w_y}^{p_y+w_y} \begin{bmatrix} \delta I I_x \\ \delta I I_y \end{bmatrix}, \tag{3.4}$$

$$v_{\text{opt}} = G^{-1} b, \tag{3.5}$$

Algorithm 7 illustrates the implementation of the equations from (3.1) to (3.5), where the function Pos() is used to ensure that a pixel is located in the image frame. The inverse of a matrix and the matrix multiplication in the algorithm are omitted in our implementation since these operations are not memory intensive and have few impact on the entire performance. The algorithm contains four loops. The first two are over all the pixels of the images and the last two are over the computation window. The bottlenecks are due to the five accesses to external memory in the innermost loop. As usual, the first implementation simply pipelines the innermost loop and uses separate interfaces for the input and output arrays.

In Algorithm 7, the five pixels of $image_0$ accessed by the innermost loop include the center pixel (defined by i, j, w_i, w_j) and four other pixels around the center pixel. When focusing only on the innermost loop, the center pixel and the right pixel can easily be reused in the following iteration. In this case, the number of accesses to external memory reduces to three instead of five.

If the next outer loop is also considered, then one or two lines can be reused by exploiting a structure known as a “line buffer”, which contains two rows of the current image. Loop unrolling could also be used in this case to further improve speed. If the

Algorithm 7 Lucas-Kanade algorithm

Require: two frames of images $image_0$ and $image_1$ **and** other coefficients
Ensure: v_{opt}

for $j = 0$ to $HEIGHT - 1$ **do**
  for $i = 0$ to $WIDTH - 1$ **do**
    $G_{2\times2} \leftarrow 0$
    $b_{2\times1} \leftarrow 0$
    for $w_j = -w_y$ to w_y **do**
      for $w_i = -w_x$ to w_x **do**
        $center \leftarrow \text{Pos}(i + w_i, j + w_j)$
        $left \leftarrow \text{Pos}(i + w_i - 1, j + w_j)$
        $right \leftarrow \text{Pos}(i + w_i + 1, j + w_j)$
        $up \leftarrow \text{Pos}(i + w_i, j + w_j - 1)$
        $down \leftarrow \text{Pos}(i + w_i, j + w_j + 1)$
        $im^0_{\text{val}} \leftarrow image_0[center]$
        $im^1_{\text{val}} \leftarrow image_1[center]$
        $\delta I \leftarrow d(im^0_{\text{val}}, im^1_{\text{val}})$
        $im^0_{\text{left}} \leftarrow image_0[left]$
        $im^0_{\text{right}} \leftarrow image_0[right]$
        $I_x \leftarrow (im^0_{\text{right}} - im^0_{\text{left}})/2$
        $im^0_{\text{up}} \leftarrow image_0[up]$
        $im^0_{\text{down}} \leftarrow image_0[down]$
        $I_y \leftarrow (im^0_{\text{down}} - im^0_{\text{up}})/2$
        $G \leftarrow G + g_{2\times2}(I_x, I_y)$
        $b \leftarrow b + f_{2\times1}(\delta I, I_x, I_y)$
      end for
    end for
    $G \leftarrow \text{inverse}(G)$
    $v_{\text{opt}}[j][i] \leftarrow G \times b$
  end for
end for

next outer loop is also considered, a large buffer can be exploited to store even more lines of the image in the on-chip memory.

For simplicity, we adopted two combined optimizations in the "On-chip memory" implementation. Firstly, copy all the pixel data located in the external memory to the on-chip BRAMs for fast access. Secondly, use two variables to store the center and right pixels as described above, in order to reuse the data and reduce the initiation interval of the innermost loop.

For the two external read-only images, the caches can be applied are the read-only caches to accelerate the algorithm. Further acceleration could also be obtained by manual post-optimizations to improve the innermost loop initiation interval, below the initial value of 5 selected by the synthesis tool. It required moving a pre-fetching operation before the innermost loop, and then using the direct retrieve() method to access the data inside the loop. Then, the initiation interval could be reduced to 1. In this case, a large enough cache behaves pretty much like a line buffer.

While real-life algorithm applications compute the optical flow on relatively large images (up to several megapixels), in this section we report RTL simulation results for small images, of 64x36 pixels, each pixel represented on 8 bits. We also report miss ratios for more realistic image and cache sizes, from functional simulation in C++. As before, the "Ext. mem." and the "On-chip mem." implementations used only off-chip and on-chip memories.

We then developed several optimized "With caches" implementations. The first one used a 64-byte one-line write-only direct-mapped cache for the output vector, a 64-byte one-line read-only direct-mapped cache for the second image (which is read once in each innermost loop iteration), and a 256-byte four-line read-only direct-mapped cache for the first image (which is read five times in each iteration). The size of the read-only direct-mapped cache used for the first image is sufficient to store three lines of the image. Hence, it acts essentially as a line buffer, but without, as usual, requiring any manual code change.

The second one doubled the line size of the two read-only direct-mapped caches with respect to the first one, thus doubling both the burst size and the cache size.

The third one used a post optimization that assumes access to consecutive addresses, as described in the previous section, with the goal to reduce the initiation interval. Note that its effectiveness, as before, is reduced by a limitation of the Vivado HLS tool, which is unable to flatten a loop inside a pipelined loop (as in the matrix multiplication case).

The performance and resource utilization of the four implementations are listed in Table 3.2. The "Ext. mem." implementation, which keeps all the images in the external memory, has a very long execution time because it accesses the external memory 1.6M times. However, this algorithm (like most computer vision, machine learning and artificial intelligence algorithms) exhibits very high levels of data reuse. In particular, each pixel of the first image is accessed many times by this algorithm. Hence, the "On-chip mem." implementation that stores all data in the on-chip memory maximizes data reuse and requires only 4.7k transfers from/to the external memory. In addition, the on-chip

Table 3.2: Performance and resource utilization for various implementations of the Lucas-Kanade algorithm.

Frame size	**64x36**, window size = 5				
Implementation	**Ext. Mem.**	**On-chip**	**Small cache**	**Large cache**	**Opt. cache**
Hit ratio (%)	—	—	99.98	99.99	99.7
Execution time (ms)	43.78	**5.636**	12.01	11.7	**9.2**
Initiation interval	5	3	5	5	1
Number of transfers	1677312	**4680**	51136	23990	**23702**
Ave. bytes per transfer	4	4.9	64	126.5	**127**
Power (W)	**0.689**	**0.693**	**1.588**	1.759	1.737
Energy (mJ)	30.16	**3.906**	18.58	20.58	**15.98**
BRAM	2	4	31	45	37
DSP	21	21	21	21	21
LUT	5888	5669	27631	45366	35254
FF	7846	7604	35376	56140	46383

memory can be accessed using two ports, so the initiation interval is reduced from 5 to 3. In summary, this implementation improves performance by about 8.5x.

Two factors improve significantly the performance of the optimizations using our caches. First of all, as in the case of bitonic sorting, the caches use bursts to increase the data size of each transfer. Second, the caches exploit the very significant amount of data reuse of this algorithm. As shown in the table, even a very small cache (comparable in size to a line buffer, which is a standard implementation for this kind of algorithms) speeds up kernel execution by 3.6x while consuming only 60% of the energy.

The miss ratio of the most frequently accessed array is only 0.02% and it required only 50k data transfers of about 64 bytes each (the ideal lower bound is about 5k transfers). The larger cache doubles the transfer size and halves the miss ratio to 0.01%. The initiation interval is reduced to 1 clock cycle for the last implementation.

The last optimization accelerates the algorithm by 4.8x and reduces energy consumption by 2x compared to the "Ext. mem." implementation. Although both cache sizes are Pareto-optimal, the smaller cache probably offers the most effective cost-performance trade-off. The miss ratios for various frame sizes, window sizes and cache configurations are reported in Figure 3.8. The lowest miss ratio can be 0.000015, leading to excellent data reuse. Even for large frames and large windows, relatively small caches can obtain a low miss ratio (around 0.1%).

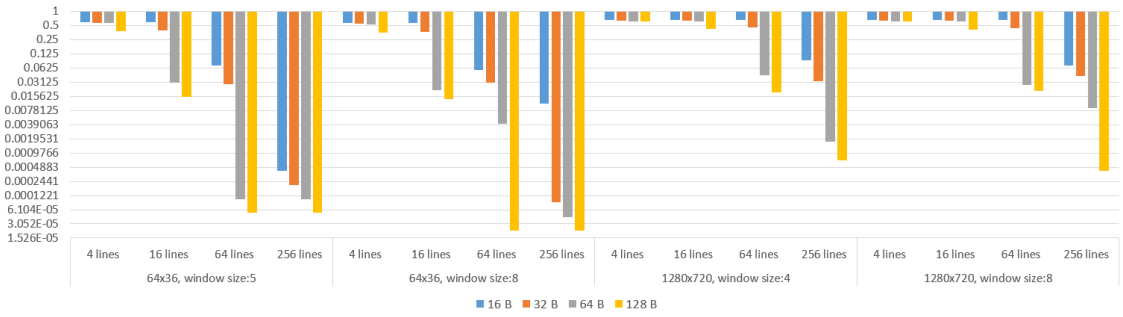

Figure 3.8: Miss ratios for different numbers of lines, data sizes and line sizes for the input image of Lucas-Kanade (log scale) .

3.3.3 Bitonic Sorting

In computer science, the sorting algorithms are among the most essential and fundamental algorithms. Various sorting schemes have been implemented in software or hardware for a large variety of applications. Bitonic sorting offers an excellent level of parallelism and it can be modified, as discussed in [41], into several phases, each of which using read-only and write-only arrays. Hence, it has also been accelerated both on FPGAs [38] and on GPUs [41].

Algorithm 8 Bitonic sorting algorithm

Require: a the array to be sorted **and** array size $N = 2^n$ **and** sorting direction dir
Ensure: $a_i \geq a_j \forall i \geq j$ for dir = true **or** $a_i \leq a_j \forall i \geq j$ for dir = false

```
for b = 1 to n do
    for s = i − 1 to 0 do
        for i = 0 to N/2 − 1 do
            dir_0 ← (i/2^(b−1))&1
            dir_0 ← dir_0 or dir
            step ← 2^s
            pos ← 2 × i − (i&(s − 1))
            a[pos], a[pos + step] ← order(a[pos], a[pos + step], dir_0) {swap two values if
            they are not in correct order}
        end for
    end for
end for
```

Algorithm 8 which contains three nested loops, is an implementation of the bitonic sorting on the CPU. The implementation of the bitonic sorting in the literature on the GPUs or FPGAs [38] are usually different due to various of optimization. In this work, we adopted the algorithm shown in Algorithm 8.

Each iteration in the outermost loop sorts blocks of size 2^b into the bitonic sequences

(i.e., sequences that are first increasing, then decreasing, then possibly increasing once more). The middle loop is over stride sizes s and is used to merge two adjacent bitonic sequences into a large sequence. The innermost loop has a constant number of iterations, and swaps the values of two data items at a distance of 2^s if they are not in the correct order.

Also in this case, the first implementation pipelines the innermost loop. Since that loop performs two read operations and two write operations to the same array in each iteration, a loop-carried dependency causes a large initiation interval, i.e., a slow pipeline throughput.

The second implementation, which is discussed, for example, in [41], divides the algorithm into two parts. The first one splits the global array into multiple arrays, each with the size equal to the on-chip memory size, and then it uses Algorithm 8 to sort these small arrays into bitonic sequences. The second part merges these bitonic sequences into the fully sorted array.

The third implementation assumes that the array to be sorted can fit in local memory, and then uses Algorithm 8 to sort it. Of course this is unrealistic for large arrays, but it has been included to show the best achievable performance.

Our cache-based implementations, due to the read and write stride accesses to external memory shown in Algorithm 8, require 2-way set-associative caches to achieve the best performance. Note that if the stride size is relatively small (smaller than the cache line size), one can easily prove [4] that the two values are stored in the same cache line after one fetching. Even if the stride size is large, the two values will be mapped to different cache lines in the same set. This guarantees the two write operations to hit.

The two read and two write operations in the innermost loop would still create a loop-carried dependency, as discussed above, and require a large pipeline initiation interval. However, one can easily note that the two write operations can never be misses because they access the same array addresses as the read operations. Thus, in this case we can use the modify() method to significantly reduce the initiation interval and dramatically improve the performance.

In order to further remove the dependency created when the two accesses conflict with each other, we can consider one more optimization. We exploit the fact that the iterations in the innermost loop are independent, hence the loop can be unrolled. Memory traces showed that once the 2-way set associative cache fetched the new data into the cache line, a number of following iterations would never miss. The number depends on the cache line size, but if this number of iterations is grouped together via partial loop unrolling, then only one initial access would need to go through the miss check, while the following unrolled iterations can just use the retrieve() and modify() methods to improve performance. This algorithm is memory-dominated and with limited

[4] Considering that the sequence starts from position 0 and that both the stride size and cache line size are the powers of 2.

data reuse. Nevertheless, without requiring almost any source code change our caches improved the performance, mostly by accessing the memory in bursts.

We performed RTL simulation of six total implementations, each sorting arrays with 128, 1024 and 4096 words filled with random integers. For the "Limited on-chip mem." implementation, which uses the limited on-chip memories to sort sub-arrays, we considered the maximum on-chip RAM sizes to be $L_{\max} = 128$ bytes and 256 bytes. Note that we have to use these small sizes, because the RTL simulation is very slow. As usual, we also report results on miss ratios for larger arrays and caches in Figure 3.9. The last

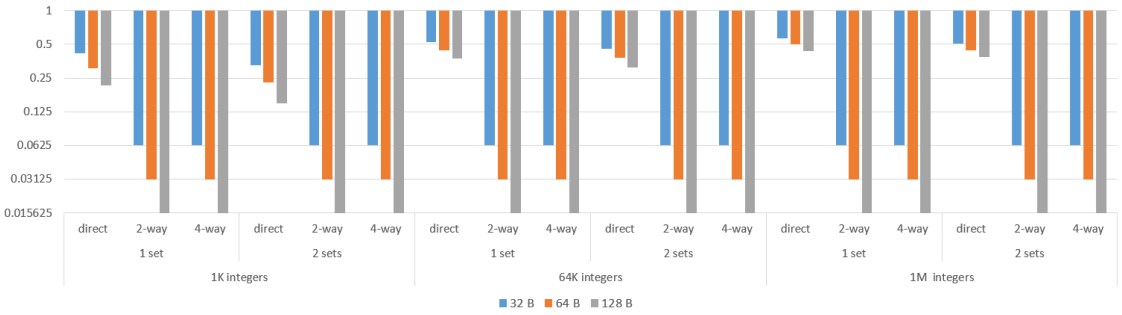

Figure 3.9: Miss ratios for different numbers of lines, data sizes and line sizes for bitonic sorting (log scale).

three "With caches" implementations were accelerated using various cache types and configurations. The first cache implementation was 2-way set-associative, with 128 total bytes and a line of 64 bytes. The second cache implementation used the same configuration but with a post-cache manual optimization, namely we replaced some write accesses with calls to the member function that assumes that the data to be written are already in cache and does not cause a flush. The third cache implementation added a manual pre-fetch loop before the array access code in the original implementation, thus avoiding the external memory loop latency in the main pipelined loop. Two configurations of the 2-way set-associative caches were implemented in order to test the effects of cache sizes on performance. One implemented a 128-byte 2-way set associative cache with a line size of 64 bytes, and the other one implemented a cache with a size of 256 bytes and a line size of 128 bytes.

Table 3.3 shows the performance of the implementations discussed above, sorting arrays with different lengths. As expected, the implementation with all data stored in external memory is the slowest. Transferring all data to very large on-chip memories has the best performance, about 20x faster. The other local memory implementation, with a limited maximum size ($L_{\max} = 128$ and 256 bytes) is much less effective and achieves a speedup of about 2x. The speedup achieved by a 2-way set-associative cache without any post optimization is about 1.5x. With the first optimization scheme, the speedup can reach 2.5x. Finally, pre-fetching achieves 8x speedup and saves about 40% energy consumption.

Table 3.3: Performance for various implementations of bitonic sorting applied to arrays with different sizes N, and using a cache line size of 64 bytes. $L_{\max}$, in bytes, is the maximum on-chip memory used (when limited).

Array size	$N = 2^7$	$N = 2^{10}$	$N = 2^{12}$
External Mem.	0.702	11.03	62.98
Limited on-chip Mem.	0.333 ($L_{\max} = 128$)	6.353 ($L_{\max} = 256$)	46.13 ($L_{\max} = 256$)
Full on-chip Mem.	**0.04**	**0.577**	**3.241**
Set-assoc. cache	0.494	7.571	42.93
1st opt. set-assoc. cache	0.287	4.473	25.36
2nd opt. set-assoc. cache	**0.0815**	**1.388**	**7.865**

Power, resource utilization and data transfer statistics for the array with size $N = 2^{10}$ are shown in Table 3.4.

Table 3.4: Performance and resource utilization of various optimizations on bitonic sorting applied to arrays with size $N = 2^{10}$.

Implementation	**Ext. Mem.**	**On-chip Mem.**		**Set-associative cache**		
		$L_{\max} = 256$	Full	Orig.	1st opt.	2nd opt.
Initiation interval	20	4		50	28	3
Exec. time (ms)	11.03	6.353	**0.577**	7.571	4.473	**1.388**
Number of transfers	84136	488414	**128**	7040	7040	**3520**
Ave. bytes per transfer	4	4.16	64	64	64	**128**
Power (W)	**0.451**	0.683	**0.465**	1.534	1.179	2.155
Energy (mJ)	4.975	4.339	**0.268**	11.61	5.274	**2.991**
BRAM	1	1	2	16	16	31
LUT	1575	11633	1441	12546	8843	22142
FF	2045	10585	1971	22669	16882	31101
DSP	0					

The first implementation keeps all data in external memory. It consumes the least

power due to its simple architecture. It performed 85k data transfers, each reading or writing only 4 bytes, since in this case the HLS tool was not able to automatically infer burst accesses.

The "On-chip mem." implementation is much faster and achieves most of its performance gains by making only 128 data transfers of 64 bytes each, in burst mode.

The caches are also able to similarly reduce the total number of transfers and increase the burst size of each access. As mentioned above, the bitonic sorting kernels from which we started had no data reuse, so the caches help only by coalescing accesses in bursts. The implementation with the 2-way set-associative cache only required 7k memory transfers, each containing 128 bytes. The bottleneck for this implementation is the initiation interval of the innermost loop, which is 2.5x larger than in the "Ext. mem." implementation and 12.5x larger than in the "On-chip mem." and the best "With caches" implementations. There are two main reasons for this long initiation interval. First, there are two read operations and two write operations in each iteration. Even though the write operations never miss, the synthesis tool is not able to ignore the false dependencies between the write-back of a dirty line and the read which updates the line, in case of a read miss. Hence, the first optimization decreases the initiation interval by around 2x, while keeping the number of transfers essentially identical, thus improving performance by about 2x.

The second optimization used twice the total cache size, halved the number of transfers and managed to achieve an initiation interval of 3 by pre-fetching the data, and hence preventing the false memory access dependencies in the main loop.

Table 3.5: Effect of cache sizes on the performance of bitonic sorting

Array size	$N = 2^7$		$N = 2^{12}$	
Cache line size (byte)	64	128	64	128
Cache size (byte)	128	256	128	256
2nd opt. set-assoc.	**Exec. time (ms)**			
	0.1288	**0.0815**	11.28	**7.865**
	Device power (W)			
	1.263	2.041	1.253	2.146
	Energy consumption (mJ)			
	0.163	0.166	14.13	16.88

Table 3.5 shows the execution time and device power required by the implementations with different line sizes for the two arrays respectively. Doubling the number of lines improves performance by about 1.5x, but also increases device power by the same factor. I.e., it improves performance and increases resource cost, but keeps total energy consumption essentially the same. As Figure 3.9 shown, the miss ratio is dramatically

reduced with a 2-way associative cache instead of a direct-mapped one. More than 2 ways or more than 1 set have no effect on the miss ratio.

Chapter 4

Acceleration of Machine Learning Algorithms

Machine learning sprouted many years ago but suffered a long AI winter. Only in the recent decade, it started to grow faster and faster. The complexity and cost of both the AI algorithms and the platforms used to execute them increase exponentially. The success of the machine learning algorithms relies on the huge amount of training data and the available high performance computing systems. A key factor of the further success of the machine learning will be wether the inference systems can meet the performance, throughput and power requirement of the applications.

4.1 Introduction

A machine learning algorithm is programmed in a very different way from a traditional algorithms. The typical method to design an algorithm depends on the mathematical models, data structures and the logics to solve a problem such as the sorting algorithm. Parallelizing such algorithms, to exploit modern architectures which rely on massive SIMD-style parallelism to provide ever increasing performance, is a formidable task. A machine learning algorithm on the other hand "programs" and improves itself by automated training from existing data. Of course, the machine learning algorithms also rely on some mathematical models such as the artificial neural networks and the application specific designed neural network architectures. But both the algorithms and the architectures can be designed once, typically to be extremely parallel and hence scalable, and then used to solve problems (i.e. learn new abilities) in a mostly automated way. Thus *programmer ability* is almost completely replaced by *data availability*. In many applications such as pattern recognition and image classification, machine learning out performs traditional algorithms by a significant amount, at equal design effort, since that effort is needed once, and then shared among a myriad of "learned" application algorithms..

The inspiration of artificial neural networks came from biological neural networks. Figure 4.1 is an example of an artificial neural network composed by multiple artificial neurons which are connected by artificial axons. The number of neurons, the connections and their weights create infinite possible networks and functionalities. The most

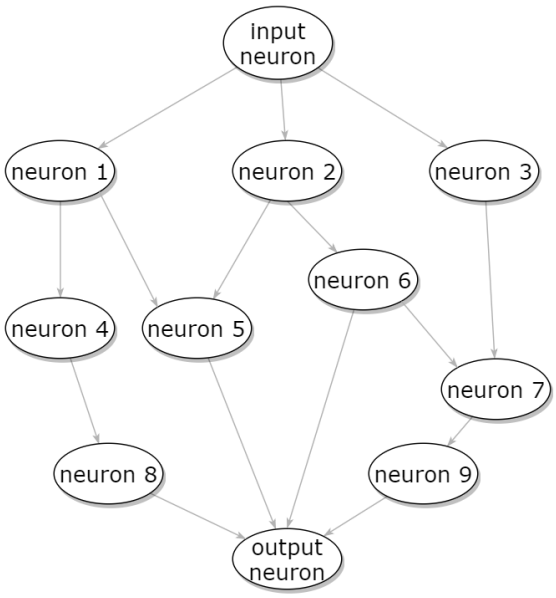

Figure 4.1: Artificial neural network example

common mathematical model for the artificial neuron is illustrated in Figure 4.2. The

neuron y gets stimuli from other neurons x_i by the linear combination of their stimuli values and then generates its own stimulus to other neurons by a non-linear function, a.k.a the rectifier and the activation function. The mathematical model is shown in (4.1), where $\mathbf{w} = (w_0, w_1, \dots w_{n-1})$, $\mathbf{x} = (x_0, x_1, \dots x_{n-1})$ and f_y is the activation function for the output stimulus of the neuron y. The behavior of an artificial neuron is then simplified as a non-linear function applied to a vector inner product.

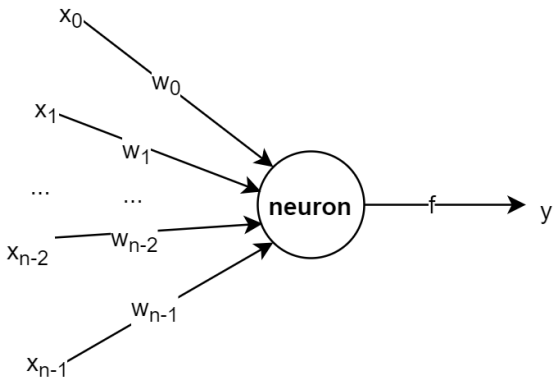

Figure 4.2: Mathematical model of the artificial neuron

$$y = f_y(\sum_{i=0}^{n-1} w_i x_i) = f_y(\mathbf{w}^{\mathrm{T}}\mathbf{x}) \tag{4.1}$$

A set of artificial neurons can be connected to form an artificial neural network. The neural network groups the neurons into layers as shown in Figure 4.3. The first layer is the input layer and the last layer is the output layer. All the other layers between the input layer and the output layer are called hidden layers. For any neuron of the layer apart from the input layer, its value is calculated by (4.1). The entire layer turns into a series of activation functions applied to a matrix-vector multiplication as shown in the (4.2), where $\mathbf{W}^l = \{w_{ij}^l\}$, $\mathbf{a}^l = \{a_i^l\}$ and $\mathbf{a}^0 = \{a_i^0\}$ is the input vector.

$$\mathbf{a}^l = \mathbf{f}^l(\mathbf{W}^l\mathbf{a}^{l-1}) \; \forall l \in \{1,2,\dots\} \tag{4.2}$$

$$\mathbf{a}^l = Relu(\mathbf{W}^l\mathbf{a}^{l-1} + \mathbf{b}^l) \; \forall l \in \{1,2,\dots\} \tag{4.3}$$

$$Relu(\mathbf{x}) = \max(\mathbf{x}, 0) \tag{4.4}$$

Many deep neural network applications adopt *Relu* (4.4) (i.e. rectified linear unit) as the activation function. So (4.2) is simplified as (4.3), where $\mathbf{b}^l = \{b_i^l\}$ is the bias vector for the layer l. Apart from *Relu*, another popular activation function is the *sigmoid function* shown in (4.5). The benefit of the *sigmoid function* is that it generates a smooth and bounded output for given inputs while its disadvantages includes the computation complexity of the exponential function and that it can cause vanishing gradient

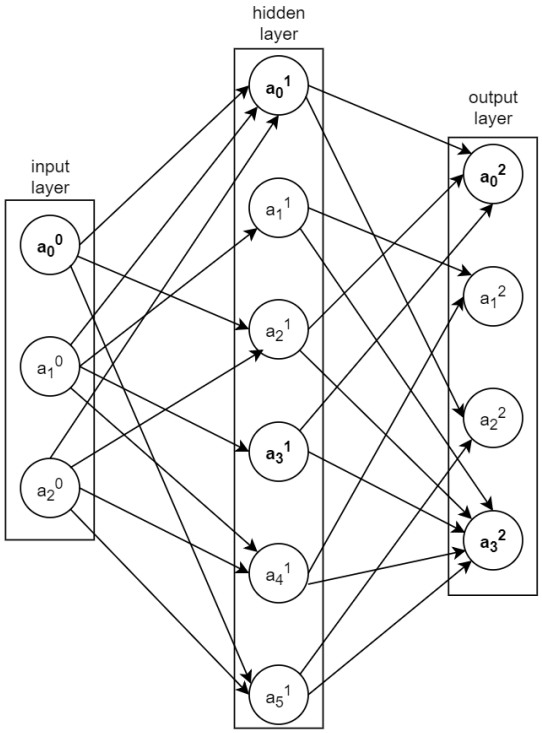

Figure 4.3: Mathematical model of the artificial neuron

problems [6, 20, 13] for deeper neural networks when the networks are trained by the gradient decending.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{4.5}$$

4.1.1 Convolutional neural network

Convolutional neural network (**CNN**) is a particular case of artificial neural network which became attractive when the AlexNet [31] won in the ImageNet competition by a large margin in 2012. Compared to its predecessor LeNet5 [32], which had been published fifteen years ago, AlexNet was designed much deeper with some special layers, and trained with more data on a much more powerful machine with GPUs. The success of the CNN is not only due to the convolution model itself, but also due to the large amount and high quality data the developers can use to train the network, and the high performance computing systems.

Convolution layer

The convolution layer is the one of the important layers of the convolution neural network due to its effectiveness in extracting features from the image data. The convolution operation with activation is modeled by (4.6) where $f^{i-1} \in \mathbb{R}^{H_{i-1} \times W_{i-1} \times C_{i-1}}$ is the

input, which is also called a set of feature maps, of a convolution layer. $f^i \in \mathbb{R}^{H_i \times W_i \times C_i}$, the output of the layer is also a set of feature maps. $b^i \in \mathbb{R}^{C_i}$, is the bias vector applied to the output feature maps. $k^i \in \mathbb{R}^{K \times K \times C_{i-1} \times C_i}$ are the kernels (i.e. weights) of the convolution layer. The relation of the input and output feature map sizes are given by $H_i = H_{i-1} - K + 1$ and $W_i = W_{i-1} - K + 1$. K is the kernel size, C_o is the number of the kernels and also the number of the output feature maps. The $ReLu$ in (4.6) can be replaced by other activation functions.

$$f^i_{h,w,c_i} = ReLu(b^i_{c_i} + \sum_{t,s \leq K, c_{i-1} \leq C_{i-1}} f^{i-1}_{h+t,w+s,c_{i-1}} k^i_{t,s,c_{i-1},c_i}) \tag{4.6}$$

Figure 4.4 demonstrates a sliding window in red box on a 7×7 feature map and its convolution (result in green box) with a 3×3 kernel. The sliding window starts from the left top corner then slides from the left to right and from top to bottom. The output is a 5×5 feature map.

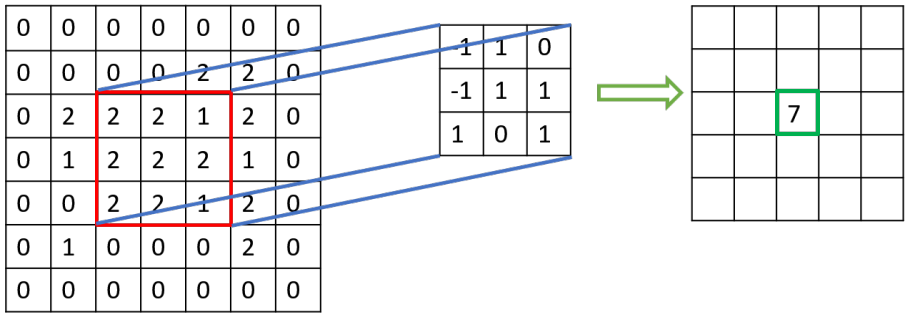

Figure 4.4: Example of a sliding window for the convolution operation

Algorithm 9 is the corresponding algorithm to compute the convolution defined in (4.6). There are in total six nested loops in the algorithm, which make it very computationally intensive, but also extremely parallelizable and the main computations are the Multiplication and ACcumulation (MAC).

Pooling layer

Each pooling layer follows one or some convolution layers aiming at summarizing information from the feature maps and reducing the sizes of the feature maps. The max pooling is the most frequently used in the CNN. As the name indicates, the output of this operation takes the maximum value from a sliding window. Usually the stride size is identical to the sliding window size. Figure 4.5 illustrates an example of max pooling layer applied to a 4×4 feature map and obtaining a 2×2 condensed feature map.

Algorithm 9 Convolution layer

Require: input feature maps $f^{i-1}[H_{i-1}][W_{i-1}][C_{i-1}]$ **and** convolution kernel $k[K][K][C_{i-1}][C_i]$ **and** bias $b[C_i]$ **and** output feature maps $f^i[H_i][W_i][C_i]$
Ensure: $Conv(f^i, f^{i-1}, k, b)$ satisfies (4.6)

```
1:  for h = 1 to H_o do
2:    for w = 1 to W_o do
3:      for c_i = 1 to C_i do
4:        tmp ← b[c_i]
5:        for t = 1 to K do
6:          for s = 1 to K do
7:            for c_{i-1} = 1 to C_{i-1} do
8:              tmp ← tmp + f^{i-1}[h + t][w + s][c_{i-1}] * k[t][s][c_{i-1}][c_i]
9:            end for
10:         end for
11:       end for
12:       f^i[h][w][c_i] ← ReLu(tmp)
13:     end for
14:   end for
15: end for
```

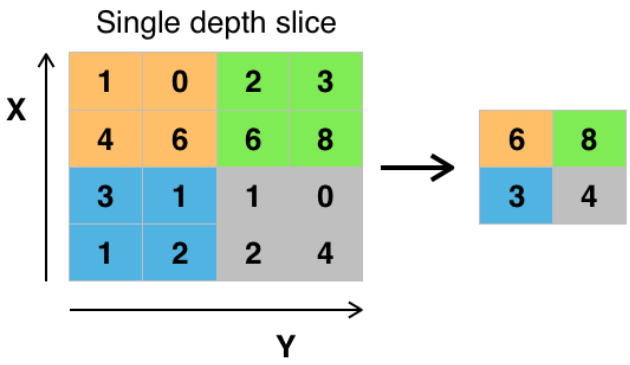

Figure 4.5: Example of the max pooling [55]

4.1.2 Neural Network on FPGAs

In this chapter, we chose several types of artificial neural networks including feed-forward neural networks (e.g. CNNs) and recurrent neural networks (e.g. LSTMs) and accelerated them on FPGAs via HLS. Modern neural networks usually have millions of weights and billions of "MAC" operations per inference, so that these models are both computation-intensive and memory-intensive. In order to accelerate neural networks on FPGAs, we need to apply comprehensive optimizations in order to achieve a good performance and energy consumption.

From the design of a neural network to its implementation on a hardware, the entire procedure involves multiple tools and programming languages. Generally speaking, neural networks are modeled and trained in machine learning design frameworks (e.g. Tensorflow, Pytorch), then optimized in high-level synthesis tools (e.g. Vivado HLS, Catapult HLS) and finally implemented on an FPGA or FPGAs, as shown in Figure 4.8. However, Tensorflow currently can generate inference code only for CPUs and GPUs, and not for HLS. Hence we first designed a tool filling this gap, to save developers from tedious code rewriting from one language to another, and to avoid mistakes due to misunderstandings among different developing teams.

Then we propose a dataflow-based acceleration strategy and an efficient implementation methodology for convolution layers based on it. We discuss several hardware architectures to accelerate a given neural network based on this methodology according to the size of the neural network and the resource limitation of the target FPGA or FPGAs.

Finally, we demonstrate the effectiveness of the methodology by applying it to two neural networks, namely ShiftShuffleNet (a CNN) and CapaNet (a RNN), by accelerating them using the dataflow-based methodology and implementing them on a PYNQ Z2 platform. The results that we obtained are very competitive with respect to the state of the art, especially on an embedded platform like the PYNQ.

Please note that in this chapter we focus on inferencing, rather than training, because so far:

- Inferencing is the most energy-intensive phase, since it is typically performed orders of magnitude more times than processing training data.
- Training requires the use of floating point arithmetic, which, as discussed in Chapter 2, is much more difficult to implement on FPGAs than integer arithmetic.

While there can be scenarios where on-line training on an FPGA can be advantageous (or even the only option, due to the lack of a fast connection to the cloud), we are leaving efficient FPGA implementation of CNN and RNN training to future work.

4.2 Design, Training and Inference Automation

4.2.1 Tensorflow

TensorFlow (denoted as **TF**) is one of the best known open-source machine learning frameworks for algorithm design, training and inference. This framework is developed and maintained by the Google brain team. It works in multiple programming environments such as python and C++. The python framewrok is the most commonly used one. **TF** uses the libraries of arithmetic operations provided by the *Numpy* package and provides many useful operations for neural networks such as *Conv* and *Relu*.

The typical design flow in **TF** starts from the construction of the neural network from the basic operations organized into a unified dataflow graph by the high level APIs [59]. Figure 4.6 is an example of the dataflow graph in **TF** for the basic layer of the neural network described by Figure 4.3 or by (4.3). The vectors and multi-dimension matrices are called tensors [1] in the **TF** environment. The weights, also tensors, are updated in the training phase. The nodes in the graph are functions such as tensor addition and multiplications. Their inputs are tensors and control signals. The edges indicate the data flows and communications among the nodes. From the computation dataflow graph, **TF** automatically generates the reversed derivative computation graph for back-propagation as shown in Figure 4.7. Then by specifying the optimizer and the loss function, **TF** can train the weights of the networks. Finally, with the trained weights, the developers can integrate the machine learning models into the applications if the results are good enough.

TF allows the user to specify the devices to execute the operation of each node in the computation graph for both training (typically performed on GPUs, which have fast double precision ALUs) and inference (often performed on FPGAs in order to save energy, both for embedded and for datacenter applications). The devices can be CPUs, GPUs or Tensor processing units (TPUs).

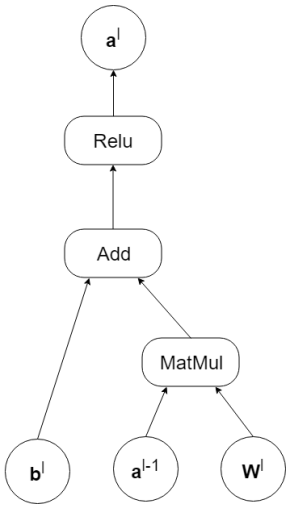

Figure 4.6: Computation graph created by **TF** for a layer of the neural network

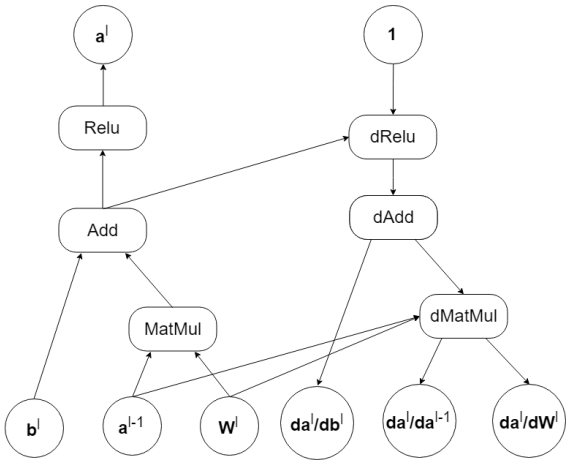

Figure 4.7: Computation graph auto-generated by **TF** for back-propagation

4.2.2 Design Flow and Code Generation

Tensorflow inference, which is the operation on which we are focusing in this chapter, typically requires the support of the tensorflow library on the target platform. This is not currently the case for FPGAs or ASICs. Hence in this chapter we discuss the so-called bare-metal implementation of a TF network, which does not assume the availability of models for TF nodes. In addition to portability to yet unsupported platforms (e.g. FPGAs), this also allows *cross-layeroptimizations*, which go beyond individual TF nodes and can be applied to a model of the full CNN represented e.g. in C++.

The neural network design and implementation flow is depicted by the Figure 4.8. The neural network is first designed and trained in **TF** and then the network pruning [23] and weights quantization are applied if applicable. Once the neural network

performance in terms of the model accuracy and the network size is acceptable, we need a bare-metal model that can be further optimized in any high-level synthesis tools. Then we could estimate the speed of the hardware accelerator using the high level synthesis tool estimate or from RTL simulation. Finally, we generate bitstream that can be implemented on the target FPGAs and we measure the performance and the energy consumption of the hardware accelerators.

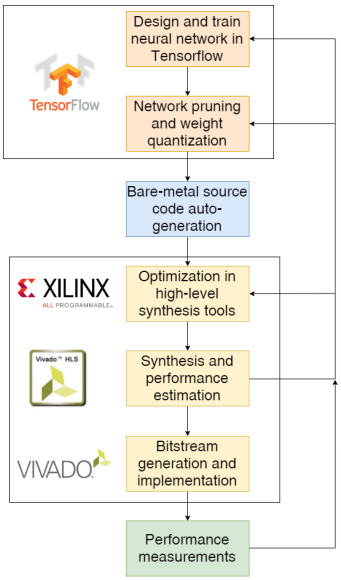

Figure 4.8: Design flow of the neural networks on an FPGA via the Tensorflow framework

One part missing in the design flow is the bare-metal source code generation algorithm that can extract both the architecture of the neural network and the trained weights from the framework automatically. So we designed a tool in python to parse the neural network, represented as a computation graph in **TF**, and convert it into a bare-metal C++ project that can be used in any high-level synthesis tool.

Handling Data Types

The first issue is to determine the form of the tensor expressions. The tensors as introduced above, can be scalar values, vectors or multi-dimensional arrays. Dynamic array allocation is not supported by state of the art high-level synthesis tools, because arrays are mapped to physical memories in the implementation. So the dimension of any tensor, including the function output, must be known at the compile time. In this project, we defined a new data type *Tensor* as a template class in C++ to store the various kinds of tensors that can be used in a **TF** network. We use template arguments to define the number of dimensions and their sizes for each tensor. This makes the object more

reusable and modular, simplifies the structure of the generated TF network code, and reduces possibilities of errors due to the weak typing rules of C++.

For some the neural networks accelerator designs on the FPGAs, the feature maps (i.e. 3D or 4D tensors) are treated as data streams. In this case, the parse of this particular tensors has different data types. In some other cases, the all the tensors may be parsed into 1D arrays rather than multi-dimension arrays. So the data type of the parser can be configured by the users.

Weight Nodes

Weights are special in the computation graph since they are the leaf nodes and usually they are extremely numerous (from hundreds of thousands to hundreds of millions for current CNNs). Given a computation graph as Figure 4.6, the weight nodes are easily found by depth-first searching. The data in each node are stored as binary bytes and can be processed by "Numpy" in python. The data type and dimension information can also parsed from the nodes. In the code generated by our parser, the constant weights are stored in a text file (to be included in the synthesized source or to be read at runtime, depending on the size of the network) and the corresponding variables are declared and initialized in a header file.

Function Nodes

Function nodes hold the type of tensor operations such as the 'matrix multiplication', 'convolution','tensor reshaping', etc. From each node, we can parse the node name, operation name, function arguments, data types and etc. For example, the matrix multiplication in **TF** is named as "matmul". The main information associated to the operation nodes is the output tensors of the function including the number of the outputs, the types and the dimension information of each tensor.

Other Nodes

There are some special nodes in the computation graph such as the "place holder" which is parsed as the input and the "identity" function which is parsed as an interface to load the weights.

4.3 Feed-Forward Neural Network

In this part, we focus on the acceleration of the feed-forward neural networks (e.g. CNN), where there is no feedback connections among layers and all the data flow in one direction layer by layer. According to the target device, the designers have to choose to accelerate the entire neural network or part of the network on each FPGA. Depending

on the computational complexity and the number of high-precision FP operations required, a givemn layer may be more efficiently executed on a CPU, an embedded GPU, or one of the FPGAs.

4.3.1 Dataflow-Based Acceleration

In this thesis, we accelerated the feed-forward neural networks based on the "Dataflow" optimization introduced in Chapter 1. The cascading of layers of the feed-forward neural network is very suitable for the dataflow-based optimization [51], especially for small networks. This type of optimization can dramatically reduce the external memory accesses, because feature maps are streamed between layers, rather than being written back and read from DRAM.

In the convolution, each output feature element requires the information of all the input feature maps located in a window whose size is the same as the kernel size. In the dataflow scheme, since all the layer blocks are connected by FIFOs with proper size and the data in the FIFOs can be read only once, we need buffers and also need to adopt special data accessing pattern of the feature maps to reduce the buffer size.

Figure 4.9 demonstrates a window from the feature maps. The height and width of window is determined by the kernel size (3 in this example). The depth of the window is the same as the number of the feature maps or the channels of the images. Hence, the best data access pattern orders pixels from left to right (along the width) and then from top to bottom (along the height) for an input with a single feature map. For an input having multiple feature maps, the access pattern moves first across all the feature maps (along the channel, e.g. channel-last format) for any position of a feature map in order to achieve the minimum buffer size. As shown in Figure 4.9, the letters from 'a' to 'h' denotes the 8 feature maps.

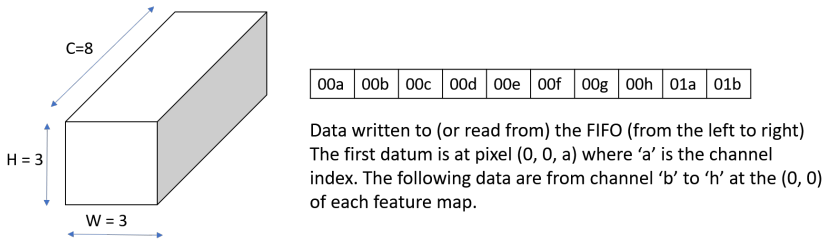

Figure 4.9: Dimension of the sliding window and data accessing pattern

Sliding Window Generation

The sliding window generation is based on the most common architecture in computer vision, namely a line buffer and a window buffer [65] as shown in Figure 4.10. The kernel size in this example is 3×3 so we need 3×3 windows to do the convolution. The feature map is shown on the right with different colors for each row. Instead of an

multi-dimension array, the feature maps are located in the input stream (FIFO). The line buffer and the 3x3 window buffer are initialized with zeros. The line buffer is used to store the pixels in the two most recently visited rows while the 3×3 window buffer stores recently visited pixels to form a 3×3 sliding window. On every clock cycle, the window buffer shifts to left and the right-most column is filled with the data from the corresponding column in the line buffer. Then a new pixel is read from the input stream and then it is stored in the line buffer (where the old pixel is no longer used, because the convolution has moved past it) and is also written in the right bottom corner of the window buffer. By using these two buffers, the sliding windows are generated with the minimum cost.

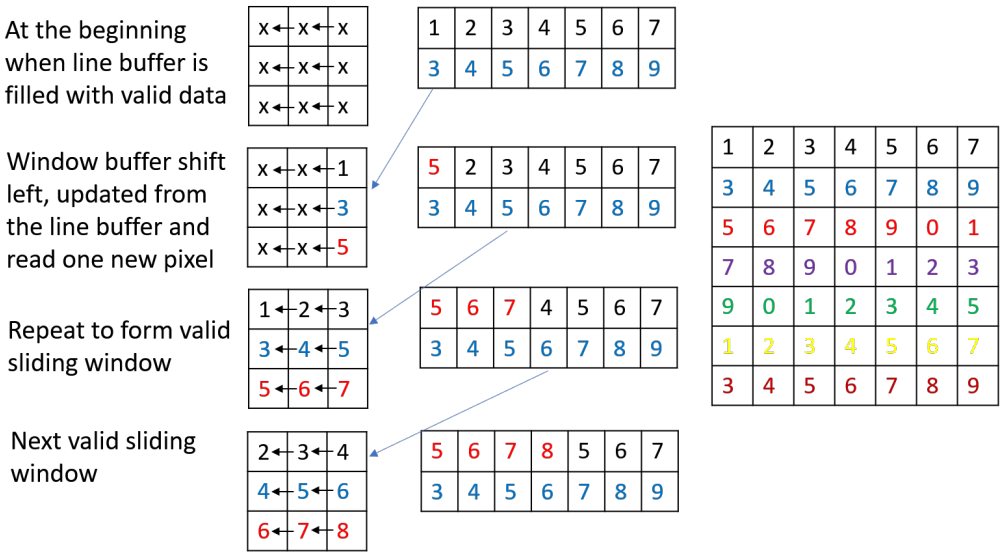

Figure 4.10: Sliding window generation

This is an example of a sliding window for a single feature map. Due to the number of input feature maps for each convolution layer and the data order in the input stream, both the line buffer and the window buffer are 3-D buffers with one dimension being as large as the number of input feature maps in use. For instance, the window buffer should have the same size as the desired sliding window, as shown in Figure 4.10. Using a set of sliding windows for all the feature maps in this order automatically guarantees the correct output stream data order.

The sliding window generation algorithm is followed by a vector inner product algorithm to compute the convolution for one output pixel.

4.3.2 Hardware Architectures on the FPGA

For different target FPGA, we need to determine the number of layers that can be accelerated on the device with the dataflow-based acceleration. The more the layers

on the device, the less the data access to the external memory. As well, we also need to consider where to store the weights on-chip or off-chip. In this section, we use the VGG16 network [43] which has 16 layers, as an example. The network architecture of the VGG16 is shown in Figure 4.11. All the layers are classified into 7 clusters. The first 6 clusters are composed by repeating convolution layers and followed by a max pooling layer.

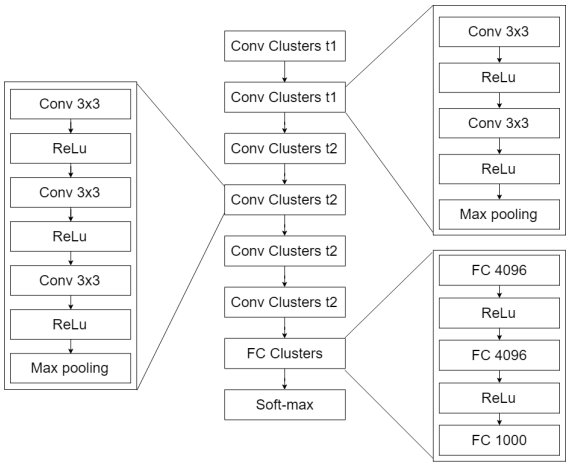

Figure 4.11: Diagram of the VGG16 net

Entire Network on Single FPGA

For a large FPGA such as the F1 FPGA, the entire VGG16 can be loaded on-chip. The best performance can be achieved if all the weights and bias are also located on-chip. However, due to the size of the weights and biases of VGG16, it is infeasible for them to be all stored in the on-chip BRAMs. During the runtime, each layer needs to fetch the weights from the external memory to on-chip BRAMs as shown in Figure 4.12.

Multi-Node System

Large neural networks that do not fit on a single FPGA, can be accelerated on a platform composed by multiple FPGAs. For a multi-node system, the layers of the network can be partitioned according to the resource and performance of each node. As shown in Figure 4.13, the 16 layers of the VGG16 are accelerated by 3 FPGAs. On AWS, all the FPGA nodes are not mutually connected so that the communication must go through the host CPUs. However, in general multi-node FPGAs potentially can communicate via an Ethernet or another serial interface. For example, Zhang et al. in [64] designed a dynamic algorithm to allocate the CNN layers on multi-FPGA system where each FPGA is connected via Xilinx Aurora protocol.

According to the size of the layers, the weights and bias can be fully-loaded on-chip or copied on demand from the off-chip memory.

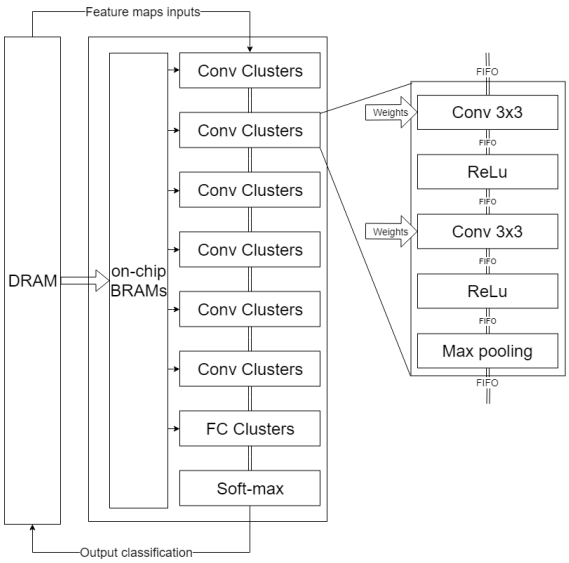

Figure 4.12: Entire VGG16 on-chip

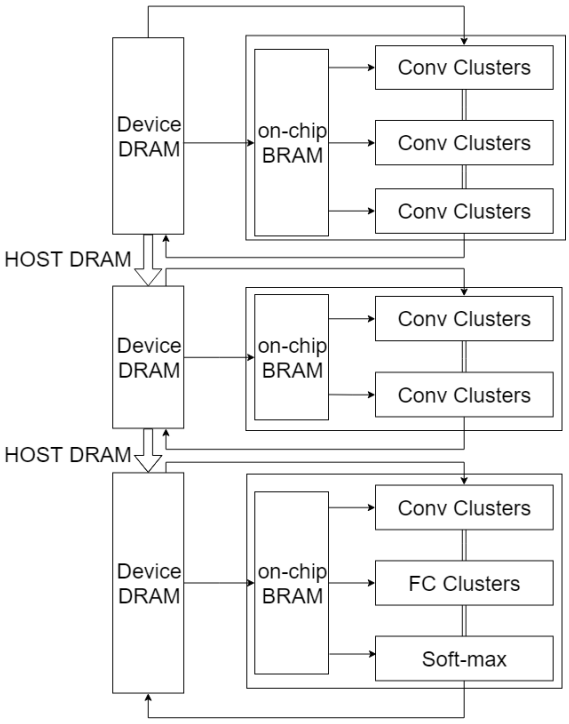

Figure 4.13: VGG16 partitioned and accelerated on multi-nodes. On AWS, F1 FPGAs can communicate only through the host DRAM control.

Partial Network on Single FPGA

It is still possible to accelerate a large neural network on a single FPGA even if all the layers do not fit simultaneously. An example hardware architecture is shown in

Figure 4.14 which has several modules such as the convolution module and FC module designed on-chip. Apart from the layer arguments, the host code also sends the configuration information to the FPGA to choose the correct datapath to accelerate the layer computation.

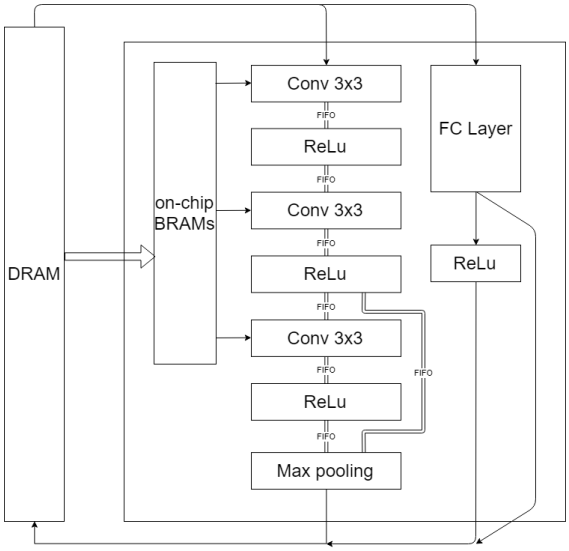

Figure 4.14: Partial layers on the FPGA to accelerate VGG16.

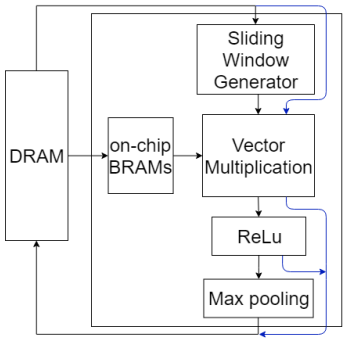

Figure 4.15: Small hardware architecture on an FPGA

For a smaller target FPGA, the hardware architecture shown in Figure 4.14 can be further tailored to meet the resource limitations, as shown in Figure 4.15, where the vector multiplication block can be used to compute the fully-connected layers or the convolution layers by using the sliding window generator discussed above.

4.3.3 ShiftShuffleNet on Embedded FPGA

ShiftShuffleNet was designed during my visit to U.C. Berkeley, based on a highly optimized CNN variant called ShiftNet which was introduced by Wu el. [60] in 2017. In

these neural networks, the computation-intensive 3x3 convolutions were replaced by so-called shift layers, which do not require any multiplication (they can be thought of as a multiplication by a kernel with only one non-zero entry, with value 1) and point-wise convolutions. The shift-block is based on the depth-wise convolution which is shown in Figure 4.16 where the kernel is in three-dimension and the output feature maps have the same number of channels as the input feature maps. Ignoring the activation and biases, (4.7) models the depthwise convolution. The depthwise convolution was introduced in [42] and has been adopted in many CNN architectures such as the mobileNet [25] and the shuffleNet [36].

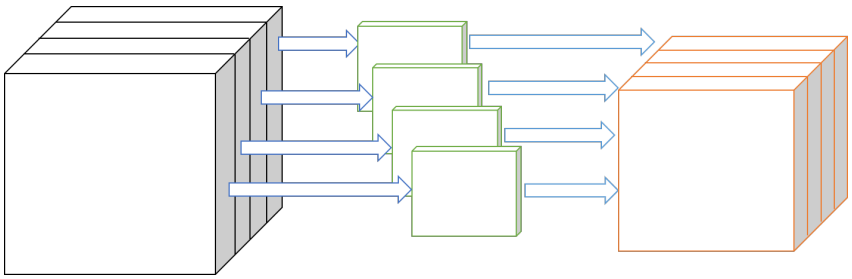

Figure 4.16: Diagram of the depthwise convolution

$$f^{i}_{h,w,c_i} = \sum_{t,s \leq K} f^{i-1}_{h+t,w+s,c_i} k^{i}_{t,s,c_i} \tag{4.7}$$

The depthwise convolution is not used alone but accompanied with the pointwise convolutions (i.e. 1×1 convolution) which is simply a linear combination of the feature maps. For instance, ShuffleNet v2 [36] is composed by a series of blocks shown in Figure 4.17. The ShuffleNet is a very efficient model whose top-1 accuracy is 69.4% on ImageNet (2% lower than VGG16), but with only 2.3M parameters (60x smaller than VGG16) and 146M FLOPs (109x smaller than VGG16).

The shiftShuffle bock was achieved by replacing the depthwise convolution layer with the shift-layer with few modifications as shown in Figure 4.18. The shift operation introduced in [60] can be treated as a special depthwise convolution with a group of one-hot 3×3 kernels (i.e. single "1" out of the 9 weights and others "0", similar as a Dirac Delta function). In Figure 4.19, the input feature map 5×5 with zero-paddings, the kernel has only single "1" and the output feature map has the same size as the input. By comparing the input and the output, it turns out that the output feature map is equal to the input feature map shifted down one line. So there in total 9 possible shifting directions of the shift operation. The implement of the shift function just requires an address shifting rather than the "MAC" in the convolution function. It potentially reduces the FLOPs in the neural network and makes the network efficient *without losing accuracy*.

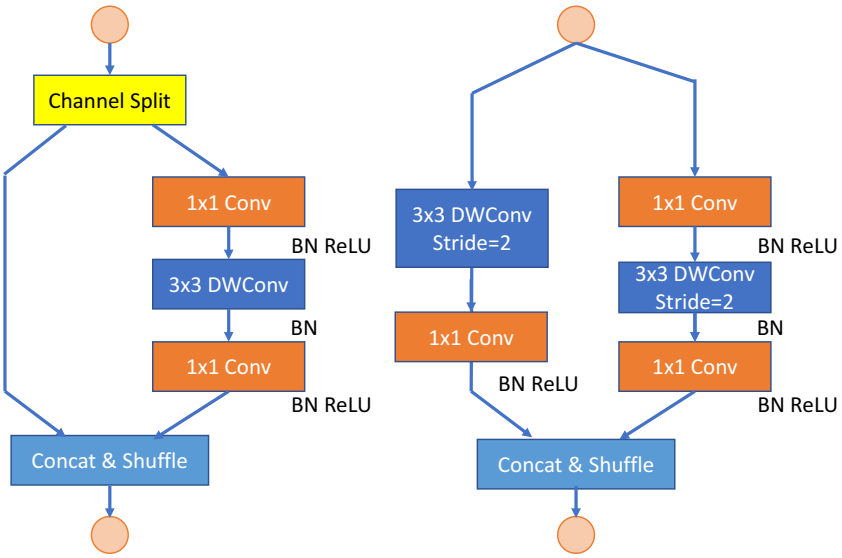

Figure 4.17: Diagram of the ShuffleNet v2 block

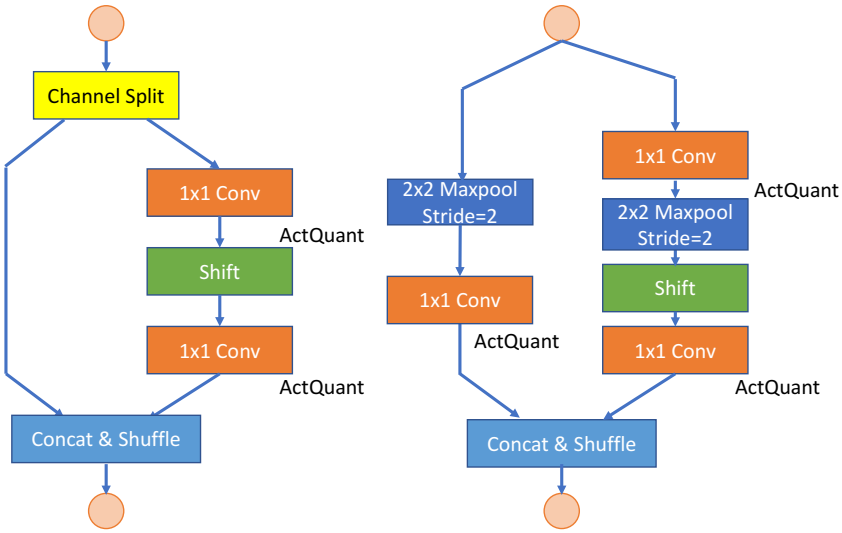

Figure 4.18: Diagram of the shiftShuffle block

Training and Quantization

The ShiftShuffleNet was designed with a macro-structure summarized in Table 4.1. This network has in total 2.9M parameters and 244M "MACs". Then the ShiftShuffleNet was quantized [62] down to various levels as shown in Table 4.2. Even with such radical quantization (1-bit for weight and 4-bit for activations), our quantized model still preserves a very competitive top-5 accuracy of 88.2%.

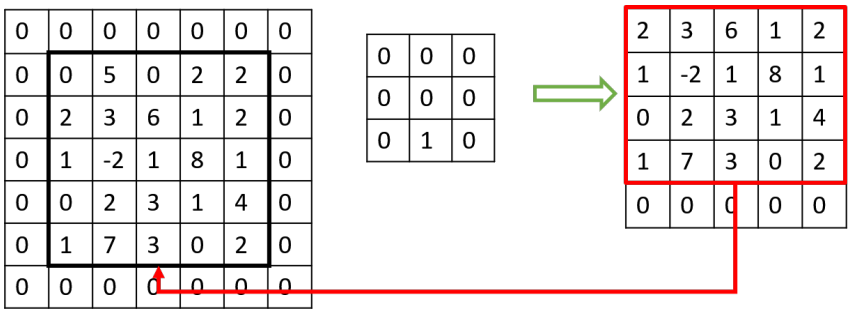

Figure 4.19: Diagram of the shift operation

Table 4.1: Macro-structure of ShiftShuffleNet

Layer	Output size	Kernel size	Stride	#Repeat	Output channel
Image	224				3
Conv1	224	1	1	1	
Maxpool	112	2	2	1	12
shift	112	3	1	1	
Conv2	112	1	1	1	
Maxpool	56	2	2	1	48
shift	56	3	1	1	
Stage 2	28		2	1	116
	28		1	3	
Stage 3	14		2	1	232
	14		1	7	
Stage 4	7		2	1	464
	7		1	3	
Conv5	7	1	1	1	1024
GlobalPool	1	7		1	1024
FC				1	1000

Accelerator Design

During my visit to U.C. Berkeley, we proposed and designed several hardware architectures to accelerate ShiftShuffleNet. One of them was published in [62]. For the sake of brevity, in this thesis we present only one of them, which is accelerated based on the "Dataflow" optimization as discussed in the previous section. The target device is the XC7Z020 FPGA on the PYNQ-Z2 platform. Since the XC7Z020 FPGA is an embedded FPGA with very few resources, it cannot load all weights to on-chip BRAMs nor accelerate all layers on-chip in a single run. The best choice is to accelerate one shiftShuffle block (i.e. a set of layers) at any given time, to select its architecture dynamically via

Table 4.2: Quantization Result of the ShiftShuffleNet, "full" stands for single precision floating point number, "w[xx]" stands for xx-bit weights and "a[xx]" stands for xx-bit activation

	full	w16a16	w8a8	w4a4	w2a4	w1a4
Top-1 Acc	69.7%	70.1%	70.3%	68.3%	68.5%	68.5%
Top-5 Acc	89.0%	89.2%	89.3%	88.1%	88.1%	88.2%

some configuration on chip with configuration options and to keep all weights off-chip.

Figure 4.20 is the hardware architecture designed to accelerate the shiftShuffle block. As shown, there are many branches on the datapath, in order to implement various kinds of layers via various choices of configuration Muxes. In Figure 4.18, there are two types of blocks in the ShiftShuffleNet. In addition, there are also several layers such as the fully connected layer are not part of the shiftShuffle block. So we designed a configurable hardware architecture on which the connections of the on-chip modules can be configured by the external controller.

In Figure 4.20, not all the layers designed on chip are equally used. For instance, the layers in green are used only few times while the blue layers are used 17 times. For this reason, we used more PEs to accelerate the blue layers. The FIFO connection in red has to be carefully sized to match the latency of the yellow layers, in order to prevent deadlocks.

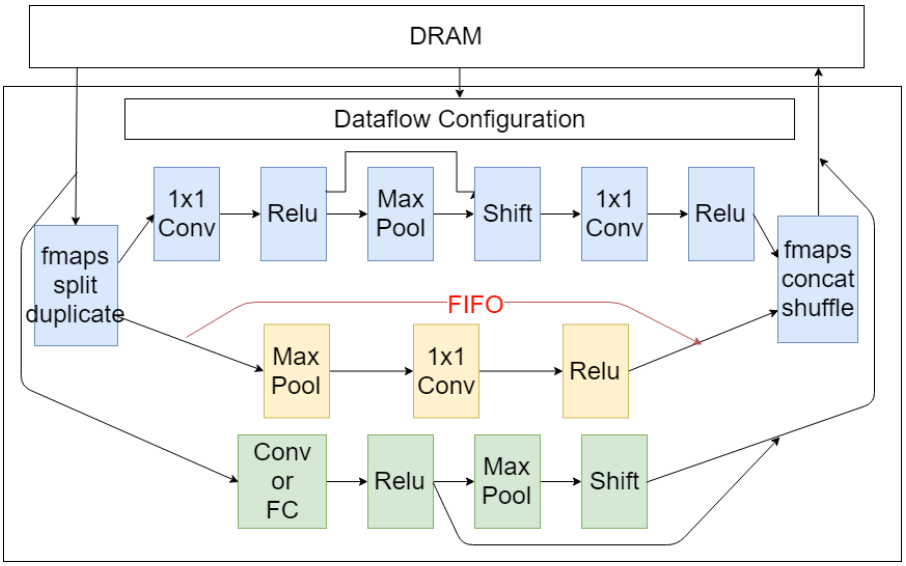

Figure 4.20: Hardware architecture of the shiftShuffle block

The shift Layer, as introduced above, can be simply performed by address shifting instead of "MACs". However, in the dataflow-based acceleration, it should generate

the data in the correct order. Since the shift function can be treated as a depthwise-convolution with special kernel configurations. So the acceleration of the shift function can refer the acceleration of the convolution function. First, generates a series sliding windows by the algorithm shown in Figure 4.10. Second, the output pixel value for each input feature map is one of the nine values in the corresponding sliding window. The value is read from the address according to the shift directions as shown in Figure 4.19. In our design, the shift direction of each feature maps is fixed. Then for each feature map, the address to be read from the sliding window is fixed.

The shift function thus avoids the vector inner product that is required to perform a traditional convolution function, and reduces the demand for computational resources (DSPs for high-precision and LUTs for low-precision weights and activations).

The ReLu Layer is implemented as a group of comparators [50] since the output value is 4-bit activation. It can be realized by a step function with 16 intervals that converts 13-bit partial sum from the convolution layer to 4-bit activation. The threshold values are different for each feature map in each layer and they are stored in on-chip BRAMs. The 16 comparators are mapped onto a binary tree structure to reduce the circuit latency (by loop unroll "directive" as introduced in Chapter 1).

Performance

In Table 4.3, we compare our accelerator against previous work targeting ConvNets for ImageNet classification with reasonable accuracy. We focused on achieving a competitive accuracy while improving the actual inference speed in terms of frames per second. The results shows that the ShiftShuffleNet achieves the highest top-1 and top-5 accuracy for the ImageNet. Compared to the first three embedded accelerators with top-1 accuracy above 60%, our model achieves the fastest inference speed up to 47 fps. An implementation with an inference speed up to 96.5 fps on a larger and faster FPGA Zynq ZU3EG is reported in [62].

Table 4.3: Performance comparison of the ShiftShuffleNet and previous works

	VGG-SVD[40]	**VGG16[44]**	**VGG16 [21]**	**DoReFa[27]**	**FINN-R [8]**	**ShiftShuffleNet**	
Platform	Zynq XC7Z045	Stratix-V	Zynq 7Z020	Zynq 7Z020	Zynq ZU3EG	Zynq 7Z020	Zynq ZU3EG
Frame rate (fps)	4.5	3.8	5.7	106.0	**200.0**	47	96.5 [62]
FPS per Watt	1.5	0.2	1.9	**46**	19.6	**23.5**	17.5
Top-1 Acc	64.64%	66.58%	67.72%	46.10%	50.3%	**68.47%**	
Top-5 Acc	86.66%	87.48%	88.06%	73.10%	N/A	**88.22%**	
Precision	16b	8-16b	8b	2b	1-2b	1-4b	
Power(W)	3.0	19.1	3.0	2.3	10.2	**2.0**	5.5
kLUTs	183	120	27	44	36	44	24
DSPs	780	760	198	89	-	117	37
BRAMs	486	1480	15	106	432	140	170
Freq. (MHz)	150	120	214	200	220	100	200

4.4 Recurrent Neural Network

Apart from the feed-forward neural networks, there is a kind of neural network with feedback i.e. a memory. This means that the output of the neural network depends not only on the current inputs but also on the past inputs and outputs, like in an IIR or FIR filter. This type of neural networks is called Recurrent Neural Network (RNN) and is used to extract information from a sequence of data, rather than from an unordered set, such in speech recognition and text translation. There are many architectures including fully-recurrent, Long Short-Term Memory (LSTM) and etc. in the RNN world.

4.4.1 Long Short Term Memory

The LSTM is a special recurrent neural network, modeled by (4.8 - 4.12), where $x_t \in \mathbb{R}^d$ is a vector sequence to be processed by the LSTM cell, $f_t \in \mathbb{R}^h$ is the forget gate's activation vector, $i_t \in \mathbb{R}^h$ is the input gate's activation vector, $o_t \in \mathbb{R}^h$ is the output gate's activation vector, $c_t \in \mathbb{R}^h$ is the cell state vector, $h_t \in \mathbb{R}^h$ is the output vector of the LSTM cell and also known as the hidden state vector, $W \in \mathbb{R}^{h \times d}$, $U \in \mathbb{R}^{h \times h}$ and $b \in \mathbb{R}^h$ are weight matrices and bias vector parameters which has to be trained. $\sigma(x)$ is the activation function defined in (4.5) and is suitable for the gate functions since its value ranges from 0 to 1 while the *tanh* is the activation function for the LSTM output vector and cell state vector.

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f) \tag{4.8}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i) \tag{4.9}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o) \tag{4.10}$$
$$c_t = f_t^T c_{t-1} + i_t^T tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{4.11}$$
$$h_t = o_t^T tanh(c_t) \tag{4.12}$$

From the equations, one can notice that there are a lot of computations which is related to both the LSTM cell size h and the input vector d. The algorithm time complexity is $O(h^2 + h \times d)$ without considering the time complexity of $exp(x)$ in both the $\sigma(x)$ and $tanh(x)$.

4.4.2 Design and Training

Experiment and Data Acquisition

The LSTM was trained in the context of another project in our group, namely, "Room locating by capacitive sensors". In this project, we used some capacitive sensors placed in a room to locate a person in the room. These sensors are assumed to be attached to any furniture or directly on the wall and are able to achieve low-cost, low-maintenance, accurate indoor location of people [47]. In the experiment used to generate the training

data for this network, we actually used a simulated set of sensors[1]. We modeled, there is a $3m \times 3m$ empty room with four sensors attached on the four walls. In order to increase the detection accuracy, sixteen infrared sensors were also placed on the ceiling of the room.

The network is used to process the data acquired from the sensors (both capacitive and infra-red) and reconstruct the location of the person

Network Design and Training

The model is based on the LSTM cell introduced above with an output layer as shown in Figure 4.21. The corresponding inference algorithm is listed in Algorithm 10. The input information is the 20 values from the capacitive sensors and the sixteen infrared sensors. The output of the network are the Cartesian coordinates of the person position in the room. We built the network in the **TF** framework by using only basic operations

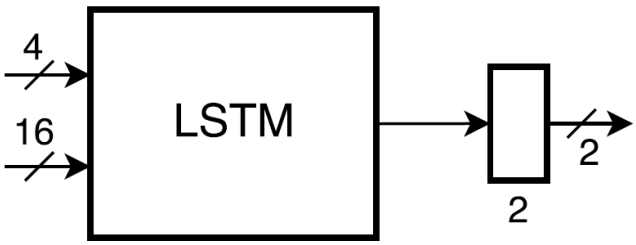

Figure 4.21: Neural network designed for sensor data processing

(e.g. matrix multiplication) in order to simplify the synthesizable C++ function node library creation (TF can instantiate also primitive LSTM cells, which we did not use). We created a Python script with several input arguments, including the LSTM cell size, the name of the file containing the training data and the corresponding coordinates, and the name of the file containing the testing data and the reference coordinates. We can also tune the training batch size, decay factor, dropout rate, and etc. in the file. The script prints the testing accuracy (mean square error) of the trained network as an output.

Table 4.4 is an example of a set of parameters used to train networks. The trained neural network then is verified by the test data set and the corresponding results are plotted in Figure 4.22. At the first few steps, the error is very high while later on, the computed path from the network is very close to the target path.

[1]Experiments to collect real data in sufficient amount are currently being carried out.

Table 4.4: Parameters used to train the network

LSTM Size	Batch size	training set size	Dropout rate
16	35	10501	20%

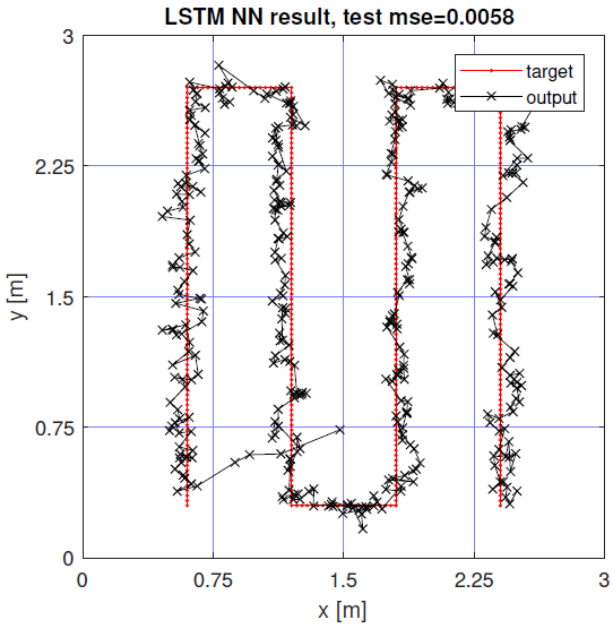

Figure 4.22: Visual output of the neural network

4.4.3 Acceleration on the embedded FPGA

Once the LSTM cell and the output layer are trained with an acceptable accuracy, we can implement the inference system on the FPGA accelerators. The inference algorithm is shown in Algorithm 10, where "matmul" is the matrix-vector multiplication. The outer-most loop is over the index of the input sequence (time sequence in this experiment). From line 6 to 14, there is the inference of the LSTM cell and line 15 is the inference of the output layer. The algorithm has multiple nested loops since the matrix-vector multiplication is realized with two loops (no tiling is needed in this small design).

For this algorithm, we focus on the following issues and optimizations:

1. This network contains about 9kB weights (4 bytes for each single precision floating point weight). Since this network is small, we can load all the weights and the bias on-chip in order to avoid the latency caused by the external memory.

2. Again, since this network is relatively small, we didn't quantize the weights. This network was accelerated on the embedded FPGA PYNQ with the data type of

Algorithm 10 LSTM inference algorithm

Require: Input vector array $x[N][I]$, LSTM size D, input array size N, input vector size I and output vector size O

Ensure: Compute output vector array $y[N][O]$ according to (4.12)

The trained LSTM cell weights $W[4D][I]$ and $U[4D][D]$, bias $b[D]$
The trained output layer weights $W_f[O][D]$ and bias $b_f[O]$
LSTM state vector $c[D] \leftarrow 0$
Hidden state vector $h[O] \leftarrow 0$
for $t = 1$ to N **do**
    Gate vector $G \leftarrow \text{matmul}(W, x[t]) + \text{matmul}(U, h) + b$
    for $d = 1$ to D **do**
        Forget gate $G_f \leftarrow G[d]$
        Input gate $G_i \leftarrow G[D + d]$
        Output gate $G_o \leftarrow G[2D + d]$
        State gate $G_c \leftarrow G[3D + d]$
        $c[d] \leftarrow \sigma(G_f)c[d] + \sigma(G_i)\tanh(G_c)$
        $h[d] \leftarrow \sigma(G_o)\tanh(c[d])$
    end for
    $y[t] \leftarrow \text{matmul}(W_f, h) + b_f$
end for

single-precision floating point numbers. In the future, we plan to design and implement a larger network with quantized weights.

3. Even though there is feedback (the hidden state of the LSTM cell) in the network rather than purely layers as in the feed-forward neural network, we can still apply the dataflow-based optimization to it. In this case, the LSTM cell is considered as a single layer and the feedback is an internal memory, within a dataflow process.

   For the matrix multiplication, we simply applied loop pipelining for output iterations and loop unrolling for input vector iterations respectively. Since the computations for each output value are independent, the pipeline could in principle achieve II=1. However, unrolling over the input vector iteration should form a balanced adder tree to minimize latency. Since the LSTM cell size can be very large, partial unroll may be used instead of full unroll. In order to limit the resource utilization in this optimization, we use a variable in the template to limit top-level parallelism, i.e. the number of PEs. In future work, when we will apply quantization, more PEs can be implemented to boost the performance.

   There are two possible implementation macro-architectures, which must be obtained from one another by code rewriting, rather than by applying HLS directives:

   - In the first one, the input vector is partitioned and the corresponding loop is unrolled. Then the outer loop can be merged with the state update loop (line 5 in the Algorithm 10). To merge the loops, we need a new array to store the new value of h and then update h in a new loop.
   - In the second one, the two matrix multiplication are synthesized as two independent processes running in parallel and driving the state update loop.

   For this particular network, both implementations achieved similar results (while this may not be the case in general, hence both need to be mentioned).

4. One important issue is the exponential function e^x which consumes a lot of DSP resources, since it needs to be fully unrolled to enable pipelining of the loops that call it, within both $\sigma(x)$ and $\tanh(x)$. In theory, to achieve the 1 LSB accuracy of both $\sigma(x)$ and $\tanh(x)$ according to their definition, we should implement $\exp(x)$ using double precision. However, we checked and even using single-precision $\exp(x)$ does not lose too much accuracy with respect to the reference C++ code (using double precision), while requiring much fewer resources than the double-precision version. In the future, we plan to use some approximation of the $\sigma(x)$ and $\tanh(x)$ functions in the network, and re-train it with the quantized weights.

   In order to reduce the resource utilization, we can also apply the directive that limits the number of the computing instances in order to force the HLS tool to share all the uses of the $\exp(x)$.

The exp(x) function can also be implemented in hardware purely by using LUTs and FFs. Vivado HLS provides directives to guide the synthesizer to implement a mathematical function with LUTs or DSPs. This type of directives was used to balance the resource utilization among DSPs, LUTs and FFs and to improve performance.

Once we completed our Design Space Exploration for this network, we were able to process all the 393 testing samples in 0.72 ms on the AWS F1 CPU. The execution time is measured by running the inference of this network 10,000 times, in order to improve the precision of the measurement. Each CPU core consumes about 8W, as reported by the AWS website. So the energy consumption is about 5.76mJ. On the embedded XC7Z020 FPGA, we used **36** PEs (with single precision floating point) to accelerate the matrix multiplication in the LSTM cell and 1 PE for the output layer. Thus in total there are 37 PEs for the network. In addition, we used **1** exp(x) instance. We achieved essentially the same performance (10%slower) than a single AWS F1 CPU. Again, this was measured by repeatedly running the 393 samples 1,000 times. The overall performance and the resource utilization on the ZYNQ board are listed in Table 4.5.

Table 4.5: Performance and resource utilization on XC7Z020 FPGA

Clock period	Execution time	Power	Energy	LUT	LUTMem	REG	BRAM	DSP
ns	ms	W	mJ	%	%	%	%	%
6	0.66	2.67	1.76	39	4	32	27	73

The relatively low performance on the FPGA accelerator (apart from the fact that it is an embedded board, while the AWS F1 CPU is a high-end Xeon processor) is due to the network size . As discussed in Chapter 1, the latency of a pipelined loop is computed by (1.3). In this example, the number of iterations of the pipelined loop is the LSTM cell size 16, i.e. it is very small compared with the pipeline depth, which is several hundreds of clock cycles due to the exp(x) function. Thus pipeline latency dominates the overall execution time, because the pipeline startup and flushing are so long that their costs are not amortized in just 16 iterations.

Chapter 5

Conclusion and Future Work

5.1 Financial Option Pricing Algorithm

Chapter 2 targets FPGA acceleration of financial option pricing models, which are computation intensive algorithms. In this chapter, we not only optimized them for implementation on both a high-end FPGA and an embedded FPGA, but also compared performance and energy consumption for three types of hardware platforms, namely, CPU, FPGA and GPU.

We explored the directive-based optimizations and the corresponding hardware architectures. We showed that high level synthesis methodologies can help the designers to obtain the desired hardware architecture easily through high level optimizations applied to a software algorithm originally implemented for CPUs or GPUs instead of time consuming RTL remodeling and optimizations.

Finally we designed a very retargetable and flexible model that can be implemented on various FPGAs such as the F1 FPGA and Z7020 FPGA. By comparing the AWS F1 FPGA and the NVIDIA P100 GPU, both are fabricated with a 16nm technology, we found that the F1 FPGA achieves about the same performance and about 4x operations per Watt than the P100 GPU. On the small embedded FPGA which is fabricated in 28nm, we achieved 23x speedup compared with the F1 CPU and about 5x operations per Watt than the GTX950 GPU which is also fabricated in 28 nm.

In our previous work [34], we also compared our results with the previous implementations both using high level and RTL designs. For the Black-Scholes model of the Asian option problem, 2.4x performance and 41.7% of energy consumption are obtained compared to a previous manual RTL design. For the Heston model of the European barrier option, this paper has achieved 2.56x of the performance and 78.1% of energy consumption of a previous implementation designed via HLS. Overall, it shows that HLS not only reduces the effort in system design, but also achieves higher performance and lower energy consumption than the traditional RTL design approach.

Since the random number generation algorithm is the bottleneck for this project due to the performance and the resource utilization of the Box Muller method, we have

several plans to replace the random number generation algorithm in the future work. For instance, the we can replace single-precision floating point data type with a fixed point one.

The Mersenne Twister algorithm also has a 64-bit implementation which is more efficient than the 32-bit version which was implemented in this project.

In the future, we also plan to implement more options such as the American option and to design a tool or script that can generate all the C++ source code for a given option pricing models with proper parameters according to the target FPGA resources and user directives.

5.2 Inline Application-Specific Caches

Chapter 3 proposed to optimize the memory-intensive algorithms by using a pre-designed C++-based inline cache rather than the tedious on-chip local memory design approach by which the designers have to care about the memory access patterns, the data synchronizations, the organizations of the BRAMs, the kernel interfaces to access the external memory and the functionality verification.

The caches designed in this project can be supported by most high-level synthesis tools and are easy to use by designers since they follow traditional cache design concept and categories such as the direct-mapped cache. We also designed several variants that can be adapted to different use contexts (e.g., read-only, write-only, etc.). They also include design aids (e.g., memory access tracing capabilities, miss ratio reporting) that can be used to ease cache size and architecture optimization.

In this thesis, we applied the cache to three algorithms from very different application areas such as machine learning, databases, and computer vision. The original algorithms with a few basic optimizations, such as loop pipelining, were used as a performance, resource usage and energy consumption reference. We also considered an ideal "best case" implementation, in which all data could fit on-chip. We then showed how using our caches, with different parameters and some further optimizations, could significantly improve performance without requiring the extensive code changes that are typically required to manually optimize on-chip memory usage. In order to fairly compare these implementations, all of them keep the same computation architecture (loop pipelining, unrolling, etc.), only changing the memory architecture. From the performance comparison results, we can conclude that the use of our caches can accelerate most memory-intensive algorithms. In summary, our cache implementations improved performance by up to 8x energy by about 2x, achieving comparable results to the best available manual optimizations of the on-chip memory architecture, while requiring a much shorter design time.

As discussed in the cache design section, the use of these inline cache has to modify the interface of the original kernel and manually configure the line size, block size and

etc. In our current work we propose a tool that is able to analyze the array access patterns, to choose a proper cache with proper configurations and to optimize the cache architectures automatically.

In the future, we will further optimize the cache to improve the performance and reduce the resource utilization. In addition, we will provide more freedoms to the users to configure the cache.

5.3 Machine Learning Algorithm

In chapter 4, we chose various machine learning algorithms as comprehensive examples to which we could applied the different types of optimization techniques for both computation-intensive and memory-intensive algorithms. This is because there is plenty of parallelism in the the machine learning algorithms especially the CNN models so that we could explore many different design points in the design space.

We first introduced a design automation tools that we developed as an HLS-oriented bare-metal code generator for the Google TensorFlow framework. With this tool, we filled the gap between **TF** framework where the neural network is trained and the high level synthesis tools where the neural network is optimized and synthesized for hardware in the design flow depicted in Figure 4.8. And then we used the tool to generate the bare-metal implementation of the YOLO net and we optimized the convolution layers by loop tiling. In the future, we could enhanse the tool to generate source code with more options around data types, e.g. the stream interfaces in the dataflow-based optimization. It could also be extended to generate the OpenCL code for X86-based platforms like the AWS F1.

Second, by exploiting the cascading architecture of the feed-forward neural networks such as the CNNs, we proposed a dataflow-based acceleration scheme which is able to reduce the external memory accesses and which can be implemented on a single FPGA or on multi-FPGA systems. According to the sizes of a given network and the resources on the target FPGAs, designers can choose one of the architectures that we propose to accelerate their neural networks.

With the dataflow-based optimization, all the layers of a network are connected by FIFOs with proper sizes. Based on the data ordering constraints, we presented an efficient sliding window generation algorithm to accelerate the key module of a CNN, namely the convolution function.

Then, we designed a neural network, called ShiftShuffleNet, which is based on the ShuffleNet and ShiftNet that were proposed in the literature, and which combines the advantages of the two networks. Our network is finally quantized with 4-bit activations and 1-bit weights, while the top-1 accuracy is still up to 68.5% and top-5 accuracy is up to 88.2%. The network accuracy is higher than those in previous work for similar (or higher) resource occupation and performance. Due to the limited resources on the target embedded FPGA, we implemented a configurable ShiftShuffle block in hardware

and then the specific dataflow paths for each layer are configured by the host processor one after the other, via a set of micro-instructions. In the end, we not only achieved a competitive accuracy but also improved the actual inference speed in terms of frames per second. In future work, on one side we can further optimize the hardware modules on the FPGA and their connections, on the other side we can redesign the neural network architecture to make it even more suitable to be accelerated in hardware.

In the last part of this chapter, we implemented a recurrent neural network, based on the LSTM model, that had been designed in the context of another project in our lab, and we used it to illustrate the methods that can be applied to accelerate recurrent neural networks. By treating the LSTM cell as a single layer, the dataflow-based optimization can also be applied to this network. In this design, we used the original datatype, single precision floating point numbers, since recurrent (i.e. cyclic) networks typically need more precision than acyclic ones. The hardware resource utilization, especially for the DSPs, is dominated by (1) the number of PEs used to accelerate the matrix multiplication and (2) the number of the exponential $\exp(x)$ instances used to compute the $\sigma(x)$ and $\tanh(x)$ functions. Finally we achieved a comparable hardware performance compared to the F1 CPU, which is much larger and much more expensive than the XC7Z020 embedded FPGA that we targeted. One important reason why the FPGA does not have much better performance than the CPU implementation is the fact that the LSTM cell size is too small compared to the pipeline depth. In the future work, we plan to implement a neural network with a larger LSTM cell, quantize the trained weights in order to avoid floating point calculations and save resources, and adopt other activation functions which are more hardware-friendly to further boost the performance and reduce the device power.

Appendix A

Direct-Mapped Cache

A.1 Code of the inline direct-mapped cache

The most significant fragments of the code of the template class of the inline direct-mapped cache are illustrated here. The `operator[]` method is overloaded and an inner class is used to differentiate between the methods to be called when the operator is used in a left-hand-side or right-hand-side context.

Listing A.1: Definition and implementation of the inline cache

```
template<typename T, int ADDR_BITS, int SET_BITS,int LINE_BITS>
class Cache {
private:

static const int CACHE_SETS = 1 << SET_BITS;
static const int LINE_SIZE = 1 <<LINE_BITS;
static const int DATA_BITS = sizeof(T) * 8;
typedef ap_uint<DATA_BITS> LocalType;

class inner {
public:
inner(Cache *cache, const int addr):cache(cache),addr(addr) {}
operator T() const{
        return cache->get(addr);
}
void operator= (T data){
        cache->set(addr, data);
}
private:
Cache *cache;
const int addr;
```

```
};

public:
typedef ap_uint<DATA_BITS*LINE_SIZE> DataType;
Cache(DataType * mem):ptr_mem(mem){...}
inner operator[](const int addr) {
        return inner(this, addr);
}
~Cache(){/* writeback code*/ ...}

private:
int requests, hits;
DataType * const ptr_mem;
DataType array[CACHE_SETS];
ap_uint<ADDR_BITS-SET_BITS-LINE_BITS>
tags[CACHE_SETS];
bool valid[CACHE_SETS], dirty[CACHE_SETS];

T get(const int addr) {
        const ap_uint<ADDR_BITS - SET_BITS-LINE_BITS> tag
        = addr >> (SET_BITS+LINE_BITS);
        const ap_uint<SET_BITS> set_i = (addr >> LINE_BITS);
        const ap_uint<LINE_BITS> block = addr;
        requests++;
        bool match = tags[set_i] == tag;

        DataType dt;
        if(valid[set_i] && match) {
                hits++;
                dt = array[set_i];
        } else {
                dt = ptr_mem[addr >> LINE_BITS];
                array[set_i] = dt;
        }
        tags[set_i] = tag;
        valid[set_i] = true;
        LocalType data = lm_data::GetData<DATA_BITS,
        DATA_BITS * LINE_SIZE,
        LINE_BITS>::get(dt, block);
        return *(T*)&data;
}
```

```
void set(const int addr, const T& data) {
        const ap_uint<ADDR_BITS - SET_BITS-LINE_BITS> tag =
        addr >> (SET_BITS+LINE_BITS);
        const ap_uint<SET_BITS> set_i = (addr >> LINE_BITS);
        const ap_uint<LINE_BITS> block = addr;
        requests++;
        bool match = tags[set_i] == tag;
        if(valid[set_i] && match) {
                hits++;
        } else {
                if(dirty[set_i]) {
                        ap_uint<ADDR_BITS> paddr=tags[set_i];
                        ptr_mem[paddr<<SET_BITS|set_i]=array[set_i];
                }
                array[set_i] = ptr_mem[addr >> LINE_BITS];
        }

        LocalType ldata = *(LocalType*)&data;
        array[set_i] = lm_data::SetData<DATA_BITS,
        DATA_BITS * LINE_SIZE,
        LINE_BITS>::
        set(array[set_i], ldata, block);
        tags[set_i] = tag;
        valid[set_i] = true;
        dirty[set_i] = true;
}};
```

A.2 Original and modified code of matrix multiplication

The basic matrix multiplication code contains three nested loops over rows, columns and inner product iteration. The innermost loop can be pipelined or unrolled as desired, by setting tool-specific directives.

Listing A.2: Matrix multiplication algorithm

```
void mat_mult(int *a, int *b, int *c) {
for (int row=0;row<rank;row++){
        for (int col=0;col<rank;col++){
                int tmp=0;
                for (int index=0;index<rank;index++) {
                #pragma HLS pipeline
```

```
                        int aIndex = row*rank + index;
                        int bIndex = index*rank + col;
                        tmp += a[aIndex] * b[bIndex];
                }
                c[row*rank + col] = tmp;
}}}
```

Listing A.3: Use CACHE in the algorithm

```
typedef Cache<int, 16, 0, a0> CacheTypeA;
typedef Cache<int, 16, b0, b1> CacheTypeB;
typedef Cache<int, 16, 0, c0> CacheTypeC;

void mat_mult(CacheTypeA::DataType *a_orig,
CacheTypeB::DataType *b_orig,
CacheTypeC::DataType *c_orig) {
CacheTypeA a(a_orig);
CacheTypeB b(b_orig);
CacheTypeC c(c_orig);

for (int row=0;row<rank;row++){
        for (int col=0;col<rank;col++){
                int tmp = 0;
                for (int index=0;index<rank;index++) {
                #pragma HLS PIPELINE
                        int aIndex = row*rank + index;
                        int bIndex = index*rank + col;
                        tmp += a[aIndex] * b[bIndex];
                }
                c[row*rank+col] = tmp;
}}}
```

Bibliography

[1] Martín Abadi, Michael Isard, and Derek G Murray. "A computational model for TensorFlow: an introduction". In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM. 2017, pp. 1–7.

[2] Michael Adler et al. "Leap scratchpads: automatic memory and cache management for reconfigurable logic". In: *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2011, pp. 25–28.

[3] Hansjörg Albrecher et al. "The little Heston trap". In: *Wilmott* 1 (2007), pp. 83–92.

[4] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[5] Julia Ankudinova and Matthias Ehrhardt. "On the numerical solution of nonlinear Black–Scholes equations". In: *Computers & Mathematics with Applications* 56.3 (2008), pp. 799–812.

[6] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[7] JM Blair, CA Edwards, and JH Johnson. "Rational Chebyshev approximations for the inverse of the error function". In: *Mathematics of Computation* 30.136 (1976), pp. 827–830.

[8] Michaela Blott et al. "FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), p. 16.

[9] Jean-Yves Bouguet. "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm". In: *Intel Corporation* 5.1-10 (2001), p. 4.

[10] Mark Broadie and Özgür Kaya. "Exact simulation of stochastic volatility and other affine jump diffusion processes". In: *Operations research* 54.2 (2006), pp. 217–231.

[11] Shaoyi Cheng et al. "Exploiting memory-level parallelism in reconfigurable accelerators". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, pp. 157–160.

[12] Jongsok Choi et al. "Impact of cache architecture and interface on performance and area of FPGA-based processor/parallel-accelerator systems". In: *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*. IEEE. 2012, pp. 17–24.

[13] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[14] Hadi Esmaeilzadeh et al. "Power challenges may end the multicore era". In: *Communications of the ACM* 56.2 (2013), pp. 93–102.

[15] Michael Fingeroff. *High-level synthesis: blue book*. Xlibris Corporation, 2010.

[16] Mark B Garman and Michael J Klass. "On the estimation of security price volatilities from historical data". In: *Journal of business* (1980), pp. 67–78.

[17] Konstantinos Georgopoulos et al. "Energy-Efficient Heterogeneous Computing at exaSCALE—ECOSCALE". In: *Hardware Accelerators in Data Centers*. Springer, 2019, pp. 199–213.

[18] Mike Giles. "Approximating the erfinv function". In: *GPU Computing Gems Jade Edition*. Elsevier, 2011, pp. 109–116.

[19] Paul Glasserman. *Monte Carlo methods in financial engineering*. Vol. 53. Springer Science & Business Media, 2013.

[20] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 2011, pp. 315–323.

[21] Kaiyuan Guo et al. "Software-Hardware Codesign for Efficient Neural Network Acceleration". In: *IEEE Micro* 37.2 (2017), pp. 18–25.

[22] John L Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[23] Song Han, Huizi Mao, and William J Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding". In: *arXiv preprint arXiv:1510.00149* (2015).

[24] Steven L Heston. "A closed-form solution for options with stochastic volatility with applications to bond and currency options". In: *The review of financial studies* 6.2 (1993), pp. 327–343.

[25] Andrew G Howard et al. "Mobilenets: Efficient convolutional neural networks for mobile vision applications". In: *arXiv preprint arXiv:1704.04861* (2017).

[26] John C Hull. *Options futures and other derivatives*. Pearson Education India, 2003.

[27] Li Jiao et al. "Accelerating low bit-width convolutional neural networks with embedded FPGA". In: *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE. 2017, pp. 1–4.

[28] Norman P Jouppi et al. "In-datacenter performance analysis of a tensor processing unit". In: *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE. 2017, pp. 1–12.

[29] Christoforos Kachris et al. "Energy-Efficient Acceleration of Spark Machine Learning Applications on FPGAs". In: *Hardware Accelerators in Data Centers*. Springer, 2019, pp. 87–107.

[30] Christian Koehler. "The Relationship between the Complexity of Financial Derivatives and Systemic Risk". In: (2011).

[31] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[32] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[33] Bruce D Lucas, Takeo Kanade, et al. "An iterative image registration technique with an application to stereo vision". In: (1981).

[34] Liang Ma, Fahad Bin Muslim, and Luciano Lavagno. "High performance and low power Monte Carlo methods to option pricing models via high level design and synthesis". In: *Modelling Symposium (EMS), 2016, European*. IEEE. 2016, pp. 157–162.

[35] Liang Ma et al. "Acceleration by Inline Cache for Memory-Intensive Algorithms on FPGA via High-Level Synthesis". In: *IEEE Access* 5 (2017), pp. 18953–18974.

[36] Ningning Ma et al. "Shufflenet v2: Practical guidelines for efficient cnn architecture design". In: *arXiv preprint arXiv:1807.11164* 5 (2018).

[37] Marvin L Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., 1967.

[38] Fahad Bin Muslim et al. "Efficient FPGA implementation of OpenCL high-performance computing applications via high-level synthesis". In: *IEEE Access* 5 (2017), pp. 2747–2762.

[39] Andrew Putnam et al. "Performance and power of cache-based reconfigurable computing". In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, pp. 395–405.

[40] Jiantao Qiu et al. "Going deeper with embedded fpga platform for convolutional neural network". In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2016, pp. 26–35.

[41] Nadathur Satish, Mark Harris, and Michael Garland. "Designing efficient sorting algorithms for manycore GPUs". In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE. 2009, pp. 1–10.

[42] Laurent Sifre and Stéphane Mallat. "Rigid-motion scattering for image classification". PhD thesis. Citeseer, 2014.

[43] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[44] Naveen Suda et al. "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks". In: *Proceedings of the 2016 International Symposium on Field-Programmable Gate Arrays*. ACM. 2016, pp. 16–25.

[45] Prasanna Sundararajan. "High performance computing using FPGAs". In: *Xilinx white paper: FPGAs* (2010), pp. 1–15.

[46] Vivienne Sze et al. "Efficient processing of deep neural networks: A tutorial and survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329.

[47] Osama Bin Tariq et al. "Performance of Machine Learning Classifiers for Indoor Person Localization With Capacitive Sensors". In: *IEEE Access* 5 (2017), pp. 12913–12926.

[48] Jürgen Teich. "Hardware/software codesign: The past, the present, and predicting the future". In: *Proceedings of the IEEE* 100.Special Centennial Issue (2012), pp. 1411–1430.

[49] David Barrie Thomas, Lee Howes, and Wayne Luk. "A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation". In: *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM. 2009, pp. 63–72.

[50] Yaman Umuroglu et al. "Finn: A framework for fast, scalable binarized neural network inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2017, pp. 65–74.

[51] Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. "Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions". In: *ACM Computing Surveys (CSUR)* 51.3 (2018), p. 56.

[52] Wikipedia, the free encyclopedia. *A plot of CPU transistor counts against dates of introduction*. 2018. URL: https://en.wikipedia.org/wiki/Moore%27s_law#/media/File:Moore%27s_Law_Transistor_Count_1971-2016.png.

[53] Wikipedia, the free encyclopedia. *An example of a Roofline model in its basic form. As the image shows, the curve consists of two platform-specific performance ceilings: the processor's peak performance and a ceiling derived from the memory bandwidth. Both axes are in logarithmic scale.* 2016. URL: https://en.wikipedia.org/wiki/Roofline_model#/media/File:Example_of_a_Roofline_model.svg.

[54] Wikipedia, the free encyclopedia. *Direct-Mapped Cache.* 2016. URL: https://en.wikipedia.org/wiki/Cache_placement_policies#/media/File:Direct-Mapped_Cache_Snehal_Img.png.

[55] Wikipedia, the free encyclopedia. *Max pooling with a 2x2 filter and stride = 2.* 2015. URL: https://en.wikipedia.org/wiki/Convolutional_neural_network#/media/File:Max_pooling.png.

[56] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Communications of the ACM* 52.4 (2009), pp. 65–76.

[57] Felix Winterstein et al. "Custom-sized caches in application-specific memory hierarchies". In: *Field Programmable Technology (FPT), 2015 International Conference on.* IEEE. 2015, pp. 144–151.

[58] Felix Winterstein et al. "MATCHUP: memory abstractions for heap manipulating programs". In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* ACM. 2015, pp. 136–145.

[59] Kanit Wongsuphasawat et al. "Visualizing dataflow graphs of deep learning models in TensorFlow". In: *IEEE transactions on visualization and computer graphics* 24.1 (2018), pp. 1–12.

[60] Bichen Wu et al. "Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions". In: *arXiv preprint arXiv:1711.08141* (2017).

[61] Ding Xie, Jimmei Lai, and Jiarong Tong. "A high utilization rate routing algorithm for modern FPGA". In: *Solid-State and Integrated-Circuit Technology, 2008. ICSICT 2008. 9th International Conference on.* IEEE. 2008, pp. 2333–2336.

[62] Yifan Yang et al. "Synetgy: Algorithm-hardware co-design for convnet accelerators on embedded fpgas". In: *arXiv preprint arXiv:1811.08634* (2018).

[63] Yu. *Overload the Brackets Operator to Perform Complex Operations.* Nov. 2014. URL: https://argcv.com/articles/3228.c.

[64] Chen Zhang et al. "Energy-efficient CNN implementation on a deeply pipelined FPGA cluster". In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design.* ACM. 2016, pp. 326–331.

[65] Yuan Zhou et al. “Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs”. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2018, pp. 269–278.

This Ph.D. thesis has been typeset by means of the TEX-system facilities. The typesetting engine was LuaLATEX. The document class was `toptesi`, by Claudio Beccari, with option `tipotesi=scudo`. This class is available in every up-to-date and complete TEX-system installation.